Pyzo

Extensions extracted

`.py`

`.html`

`.css`

Project: Pyzo

Table of Contents

Pyzo

Pyzo\doc

Pyzo\freeze

Pyzo\pyzo

Pyzo\pyzo\codeeditor

Pyzo\pyzo\codeeditor\extensions

Pyzo\pyzo\codeeditor\parsers

Pyzo\pyzo\core

Pyzo\pyzo\pyzokernel

Pyzo\pyzo\qt

Pyzo\pyzo\qt_patch

Pyzo\pyzo\resources

Pyzo\pyzo\tools

Pyzo\pyzo\tools\pyzoFileBrowser

Pyzo\pyzo\util

Pyzo\pyzo\util\interpreters

Pyzo\pyzo\yoton

Pyzo\pyzo\yoton\channels

Pyzo\pyzo\yoton\tests

Pyzo\tests

```
"""
Use Github API to get download counts of the Pyzo binaries.
"""
## Download the data
import requests
api_url = "https://api.github.com"
pyzo_api_url = api_url + "/repos/pyzo/pyzo"
response = requests.get(pyzo_api_url + "/releases")
assert response.status_code == 200
d = response.json()
## Process and display
alltime_count = 0
for release in reversed(d):
    print("{} @ {}".format(release["name"], release["created_at"]))
    total_count = 0
    for asset in release["assets"]:
        count = asset["download_count"]
        total_count += count
        print("    {}: {}".format(asset["name"], count))
    print("    Total: {}\n".format(total_count))
    alltime_count += total_count
print("    Alltime download count: {}".format(alltime_count))
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the (new) BSD License.
# The full license can be found in 'license.txt'.
""" pyzolauncher.py script
This is a script used to startup Pyzo. Added for convenience, and also
for running a test on the source version.
Pyzo can be installed as a package, but it does not have to. You can
start Pyzo in a few different ways:
  * execute this script (pyzolauncher.py)
  * execute the pyzo directory (Python will seek out pyzo/__main__.py)
  * execute the pyzo package ("python -m pyzo")
Only in the latter must Pyzo be installed.
"""
import os
import sys
import subprocess
# faulthandler helps debugging hard crashes, it is included in py3.3
try:
    if sys.executable.lower().endswith("pythonw.exe"):
        raise ImportError("Dont use faulthandler in pythonw.exe")
    import faulthandler
    faulthandler.enable()
except ImportError:
    pass
if "--test" in sys.argv:
    # Prepare log file
    logfilename = os.path.abspath(os.path.join(__file__, "..", "log.txt"))
    with open(logfilename, "wt") as f:
        f.write("")
    # Run Pyzo
    os.environ["PYZO_LOG"] = logfilename
    subprocess.run([sys.executable, "pyzo", "--test"])
else:
    import pyzo
    pyzo.start()
```

```
""" Setup script for the Pyzo package.
Notes on how to do a release. Mostly for my own convenience:
* Write release notes
* Bump `__version__`
* `git tag vx.y.z`
* `git push vx.y.z` (makes Azure build the binaries and push to a GH release)
* Update links on Pyzo website
* `python setup.py sdist upload`
"""
import os
import sys
try:
    from setuptools import find_packages, setup
except ImportError:
    from distutils.core import setup
def get_version_and_doc(filename):
    NS = dict(__version__="", __doc__="")
    docStatus = 0  # Not started, in progress, done
    for line in open(filename, "rb").read().decode().splitlines():
        if line.startswith("__version__"):
            exec(line.strip(), NS, NS)
        elif line.startswith('"""'):
            if docStatus == 0:
                docStatus = 1
                line = line.lstrip('"')
            elif docStatus == 1:
                docStatus = 2
        if docStatus == 1:
            NS["__doc__"] += line.rstrip() + "\n"
    if not NS["__version__"]:
        raise RuntimeError("Could not find __version__")
    return NS["__version__"], NS["__doc__"]
version, doc = get_version_and_doc(
    os.path.join(os.path.dirname(__file__), "pyzo", "__init__.py")
)
setup(
    name="pyzo",
    version=version,
    description="the Python IDE for scientific computing",
    long_description=doc,
    author="Almar Klein",
    author_email="almar.klein@gmail.com",
    license="2-Clause BSD",
    url="https://pyzo.org",
    keywords="Python interactive IDE Qt science computing",
    platforms="any",
    provides=["pyzo"],
    python_requires=">=3.5.0",
    install_requires=[
        "packaging"
    ],  # and 'PySide2' or 'PyQt5' (less sure about PySide/PyQt4)
    packages=find_packages(exclude=["tests", "tests.*"]),
    package_dir={"pyzo": "pyzo"},
    package_data={
        "pyzo": [
            "resources/*.*",
            "resources/icons/*.*",
            "resources/appicons/*.*",
            "resources/images/*.*",
            "resources/fonts/*.*",
            "resources/themes/*.*",
            "resources/translations/*.*",
        ]
    },
    data_files=[
        ("", ["README.md", "LICENSE.md", "pyzo.appdata.xml", "pyzolauncher.py"])
    ],
    zip_safe=False,
    classifiers=[
        "Development Status :: 5 - Production/Stable",
        "Intended Audience :: Science/Research",
        "Intended Audience :: Education",
        "Intended Audience :: Developers",
        "Topic :: Scientific/Engineering",
        "Topic :: Software Development",
        "License :: OSI Approved :: BSD License",
        "Operating System :: MacOS :: MacOS X",
        "Operating System :: Microsoft :: Windows",
        "Operating System :: POSIX",
        "Programming Language :: Python :: 3",
        "Programming Language :: Python :: 3.5",
        "Programming Language :: Python :: 3.6",
        "Programming Language :: Python :: 3.7",
        "Programming Language :: Python :: 3.8",
    ],
    entry_points={
        "console_scripts": [
            "pyzo = pyzo.__main__:main",
        ],
    },
)
# Post processing:
# Install appdata.xml on Linux if we are installing in the system Python
if sys.platform.startswith("linux") and sys.prefix.startswith("/usr"):
    if len(sys.argv) >= 2 and sys.argv[1] == "install":
        fname = "pyzo.appdata.xml"
        filename1 = os.path.join(os.path.dirname(__file__), fname)
        filename2 = os.path.join("/usr/share/metainfo", fname)
        try:
            bb = open(filename1, "rb").read()
            open(filename2, "wb").write(bb)
        except PermissionError:
            pass  # No sudo, no need to warn
        except Exception as err:
            print("Could not install %s: %s" % (fname, str(err)))
        else:
            print("Installed %s" % fname)
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
# Pyzo documentation build configuration file, created by
# sphinx-quickstart on Sun Jan 26 00:33:41 2014.
#
# This file is execfile()d with the current directory set to its containing dir.
#
# Note that not all possible configuration values are present in this
# autogenerated file.
#
# All configuration values have a default; values that are commented out
# serve to show the default.
import sys, os  # noqa
# If extensions (or modules to document with autodoc) are in another directory,
# add these directories to sys.path here. If the directory is relative to the
# documentation root, use os.path.abspath to make it absolute, like shown here.
# sys.path.insert(0, os.path.abspath('.'))
# -- General configuration -----------------------------------------------------
# If your documentation needs a minimal Sphinx version, state it here.
# needs_sphinx = '1.0'
# Add any Sphinx extension module names here, as strings. They can be extensions
# coming with Sphinx (named 'sphinx.ext.*') or your custom ones.
extensions = ["sphinx.ext.autodoc"]
# Add any paths that contain templates here, relative to this directory.
templates_path = ["_templates"]
# The suffix of source filenames.
source_suffix = ".rst"
# The encoding of source files.
# source_encoding = 'utf-8-sig'
# The master toctree document.
master_doc = "index"
# General information about the project.
project = "Pyzo"
copyright = "2016, Pyzo contributors"
# The version info for the project you're documenting, acts as replacement for
# |version| and |release|, also used in various other places throughout the
# built documents.
#
# The short X.Y version.
version = "3.4"
# The full version, including alpha/beta/rc tags.
release = "3.4"
```

```
# The language for content autogenerated by Sphinx. Refer to documentation
# for a list of supported languages.
# language = None
# There are two options for replacing |today|: either, you set today to some
# non-false value, then it is used:
# today = ''
# Else, today_fmt is used as the format for a strftime call.
# today_fmt = '%B %d, %Y'
# List of patterns, relative to source directory, that match files and
# directories to ignore when looking for source files.
exclude_patterns = ["_build"]
# The reST default role (used for this markup: `text`) to use for all documents.
# default_role = None
# If true, '()' will be appended to :func: etc. cross-reference text.
# add_function_parentheses = True
# If true, the current module name will be prepended to all description
# unit titles (such as .. function::).
# add_module_names = True
# If true, sectionauthor and moduleauthor directives will be shown in the
# output. They are ignored by default.
# show_authors = False
# The name of the Pygments (syntax highlighting) style to use.
pygments_style = "sphinx"
# A list of ignored prefixes for module index sorting.
# modindex_common_prefix = []
# If true, keep warnings as "system message" paragraphs in the built documents.
# keep_warnings = False
# -- Options for HTML output ---------------------------------------------------
# The theme to use for HTML and HTML Help pages.  See the documentation for
# a list of builtin themes.
html_theme = "default"
# Theme options are theme-specific and customize the look and feel of a theme
# further.  For a list of options available for each theme, see the
# documentation.
# html_theme_options = {}
# Add any paths that contain custom themes here, relative to this directory.
# html_theme_path = []
# The name for this set of Sphinx documents.  If None, it defaults to
# "<project> v<release> documentation".
# html_title = None
# A shorter title for the navigation bar.  Default is the same as html_title.
# html_short_title = None
# The name of an image file (relative to this directory) to place at the top
# of the sidebar.
# html_logo = None
# The name of an image file (within the static path) to use as favicon of the
# docs.  This file should be a Windows icon file (.ico) being 16x16 or 32x32
# pixels large.
# html_favicon = None
# Add any paths that contain custom static files (such as style sheets) here,
# relative to this directory. They are copied after the builtin static files,
# so a file named "default.css" will overwrite the builtin "default.css".
html_static_path = ["_static"]
# If not '', a 'Last updated on:' timestamp is inserted at every page bottom,
# using the given strftime format.
# html_last_updated_fmt = '%b %d, %Y'
# If true, SmartyPants will be used to convert quotes and dashes to
# typographically correct entities.
# html_use_smartypants = True
# Custom sidebar templates, maps document names to template names.
# html_sidebars = {}
# Additional templates that should be rendered to pages, maps page names to
# template names.
# html_additional_pages = {}
# If false, no module index is generated.
# html_domain_indices = True
# If false, no index is generated.
# html_use_index = True
# If true, the index is split into individual pages for each letter.
# html_split_index = False
# If true, links to the reST sources are added to the pages.
# html_show_sourcelink = True
# If true, "Created using Sphinx" is shown in the HTML footer. Default is True.
# html_show_sphinx = True
# If true, "(C) Copyright ..." is shown in the HTML footer. Default is True.
# html_show_copyright = True
# If true, an OpenSearch description file will be output, and all pages will
# contain a <link> tag referring to it.  The value of this option must be the
# base URL from which the finished HTML is served.
# html_use_opensearch = ''
# This is the file name suffix for HTML files (e.g. ".xhtml").
# html_file_suffix = None
# Output file base name for HTML help builder.
htmlhelp_basename = "Pyzodoc"
# -- Options for LaTeX output ---------------------------------------------------
latex_elements = {
    # The paper size ('letterpaper' or 'a4paper').
    #'papersize': 'letterpaper',
    # The font size ('10pt', '11pt' or '12pt').
    #'pointsize': '10pt',
    # Additional stuff for the LaTeX preamble.
    #'preamble': '',
}
# Grouping the document tree into LaTeX files. List of tuples
# (source start file, target name, title, author, documentclass [howto/manual]).
latex_documents = [
    ("index", "pyzo.tex", "Pyzo Documentation", "Pyzo contributors", "manual"),
]
# The name of an image file (relative to this directory) to place at the top of
# the title page.
# latex_logo = None
# For "manual" documents, if this is true, then toplevel headings are parts,
# not chapters.
# latex_use_parts = False
# If true, show page references after internal links.
# latex_show_pagerefs = False
# If true, show URL addresses after external links.
# latex_show_urls = False
# Documents to append as an appendix to all manuals.
# latex_appendices = []
# If false, no module index is generated.
# latex_domain_indices = True
# -- Options for manual page output ---------------------------------------------
# One entry per manual page. List of tuples
# (source start file, name, description, authors, manual section).
man_pages = [("index", "pyzo", "Pyzo Documentation", ["Pyzo contributors"], 1)]
# If true, show URL addresses after external links.
# man_show_urls = False
# -- Options for Texinfo output -------------------------------------------------
# Grouping the document tree into Texinfo files. List of tuples
# (source start file, target name, title, author,
#  dir menu entry, description, category)
texinfo_documents = [
    (
        "index",
        "Pyzo",
        "Pyzo Documentation",
        "Pyzo contributors",
        "Pyzo",
        "One line description of project.",
        "Miscellaneous",
    ),
]
# Documents to append as an appendix to all manuals.
# texinfo_appendices = []
# If false, no module index is generated.
# texinfo_domain_indices = True
# How to display URL addresses: 'footnote', 'no', or 'inline'.
# texinfo_show_urls = 'footnote'
# If true, do not generate a @detailmenu in the "Top" node's menu.
# texinfo_no_detailmenu = False
```

```
import os
import sys
import time
import platform
import traceback
import importlib
import dialite
TESTING = "--test" in sys.argv
# %% Utils
def write(*msg):
    print(*msg)
    if os.getenv("PYZO_LOG", ""):
        with open(os.getenv("PYZO_LOG"), "at") as f:
            f.write(" ".join(msg) + "\n")
class SourceImporter:
    def __init__(self, dir):
        self.module_names = set()
        for name in os.listdir(dir):
            fullname = os.path.join(dir, name)
            if name.endswith(".py"):
                self.module_names.add(name)
            elif os.path.isdir(fullname):
                if os.path.isfile(os.path.join(fullname, "__init__.py")):
                    self.module_names.add(name)
    def find_spec(self, fullname, path, target=None):
        if fullname.split(".")[0] in self.module_names:
            return sys.meta_path[1].find_spec(fullname, path, target)
        else:
            return None
def error_handler(cls, err, tb, action=""):
    title = "Application aborted"
    if action:
        title += f" while {action}"
    msg = f"{cls.__name__}: {err}"
    # Try writing traceback to stderr
    try:
        tb_info = "".join(traceback.format_list(traceback.extract_tb(tb)))
        write(f"{title}\n{msg}\n{tb_info}")
    except Exception:
        pass
    # Use dialite to show error in modal window
    if not TESTING:
        dialite.fail(title, msg)
```

```
class BootAction:
    def __init__(self, action):
        self._action = action
        try:
            write(action)
        except Exception:
            pass
    def __enter__(self):
        return self
    def __exit__(self, cls, err, tb):
        if err:
            if not isinstance(err, SystemExit) or err.code:
                error_handler(cls, err, tb, self._action)
                if not isinstance(err, SystemExit):
                    sys.exit(1)
# %% Boot
if TESTING:
    dt = time.strftime("%Y-%m-%d %H:%M:%S", time.gmtime())
    write(f"Testing Pyzo binary ({dt} UTC)")
    write(platform.platform())
    write(sys.version)
with BootAction("Setting up source importer"):
    if sys._MEIPASS.strip("/").endswith(".app/Contents/MacOS"):
        # Note: it looks like the source dir IS available just after we froze
it,
        # so the test_frozen run passes. However, the packaged versions only
        # have the source dir in Contents/Resources.
        # Not sure why, maybe related to symlinks?
        source_dir = os.path.join(sys._MEIPASS, "..", "Resources", "source")
    else:
        source_dir = os.path.join(sys._MEIPASS, "source")
    source_dir = os.path.abspath(source_dir)
    if TESTING:
        write(f"Source dir: {source_dir} {os.path.isdir(source_dir)}")
    sys.path.insert(0, source_dir)
    sys.meta_path.insert(0, SourceImporter(source_dir))
with BootAction("Applying pre-import Qt tweaks"):
    importlib.import_module("pyzo.pre_qt_import")
with BootAction("Importing Qt"):
    QtCore = importlib.import_module("pyzo.qt." + "QtCore")
    QtGui = importlib.import_module("pyzo.qt." + "QtGui")
    QtWidgets = importlib.import_module("pyzo.qt." + "QtWidgets")
with BootAction("Running Pyzo"):
    pyzo = importlib.import_module("pyzo")
    write(f"Pyzo {pyzo.__version__}")
    pyzo.start()
    write("Stopped")  # may be written to log twice because Pyzo defers stdout
```

```
#!/usr/bin/env python3
""" PyInstaller script
"""
import os
import sys
import shutil
from distutils.sysconfig import get_python_lib
# Definitions
name = "pyzo"
qt_api = os.getenv("PYZO_QT_API", "PySide6")
this_dir = os.path.abspath(os.path.dirname(__file__)) + "/"
exe_script = this_dir + "boot.py"
dist_dir = this_dir + "dist/"
icon_file = os.path.abspath(
    os.path.join(this_dir, "..", "pyzo", "resources", "appicons",
"pyzologo.ico")
)
# Run the script from the freeze dir
os.chdir(this_dir)
## Utils
def get_pyzo_version():
    """Get Pyzo's version."""
    filename = os.path.join(dist_dir, "..", "..", "pyzo", "__init__.py")
    NS = {}
    for line in open(filename, "rb").read().decode().splitlines():
        if line.startswith("__version__"):
            exec(line.strip(), NS, NS)
    if not NS.get("__version__", 0):
        raise RuntimeError("Could not find __version__")
    return NS["__version__"]
def _find_modules(root, extensions, skip, parent=""):
    """Yield all modules and packages and their submodules and subpackages found
at `root`.
    Nested folders that do _not_ contain an __init__.py file are assumed to also
be on sys.path.
    `extensions` should be a set of allowed file extensions (without the .).
`skip` should be
    a set of file or folder names to skip. The `parent` argument is for internal
use only.
    """
    for filename in os.listdir(root):
        if filename.startswith("_"):
            continue
        if filename in skip:
            continue
        path = os.path.join(root, filename)
        if os.path.isdir(path):
            if filename.isidentifier() and os.path.exists(
                os.path.join(path, "__init__.py")
            ):
                if parent:
                    packageName = parent + "." + filename
                else:
                    packageName = filename
                for module in _find_modules(path, extensions, skip,
packageName):
                    yield module
            elif not parent:
                for module in _find_modules(path, extensions, skip, ""):
                    yield module
        elif "." in filename:
            moduleName, ext = filename.split(".", 1)
            if ext in extensions and moduleName.isidentifier():
                if parent and moduleName == "__init__":
                    yield parent
                elif parent:
                    yield parent + "." + moduleName
                else:
                    yield moduleName
def get_stdlib_modules():
    """Return a list of all module names that are part of the Python Standard
Library."""
    stdlib_path = get_python_lib(standard_lib=True)
    extensions = {"py", "so", "dll", "pyd"}
    skip = {
        "site-packages",  # not stdlib
        "idlelib",  # irrelevant for us
        "lib2to3",  # irrelevant for us
        "test",  # irrelevant for us
        "turtledemo",  # irrelevant for us
        "tkinter",  # not needed
        "tk",
        "tcl",
        "unittest",  # not needed
        "distutils",  # not needed - also must be avoided*
    }
    # On distutils: one distutils submodule causes IPython to be included,
    # and with that, a sloth of libs like matplotlib and multiple Qt libs.
    return list(_find_modules(stdlib_path, extensions, skip))
# All known Qt toolkits, excluded the one we will use
other_qt_kits = {"PySide", "PySide2", "PySide6", "PyQt4", "PyQt5", "PyQt6"}
other_qt_kits.remove(qt_api)
## Includes and excludes
# We don't really make use of PySides detection mechanism, but instead specify
# explicitly what our binaries need. We include almost the whole stdlib, and
# a small subset of Qt. This way, future versions of Pyzo can work in the same
# container, and we still have a relatively small footprint.
includes = []
excludes = []
# Include almost all stdlib modules
includes += get_stdlib_modules()
# Include a few 3d party packages, e.g. deps of qtpy
includes += open(os.path.join(this_dir, "frozen_libs.txt"), "rt").read().split()
# Include a subset of Qt modules
qt_includes = [
    "QtCore",  # Standard
    "QtGui",  # Standard
    "QtWidgets",  # Standard
    "QtHelp",  # For docs
    "QtPrintSupport",  # For PDF export
    "QtOpenGLWidgets",  # Because qtpy imports QOpenGLQWidget into QtWidgets
]
includes += [f"{qt_api}.{sub}" for sub in qt_includes]
# There is a tendency to include tk modules
excludes += ["tkinter", "tk", "tcl"]
# Also exclude other Qt toolkits just to be sure
excludes += list(other_qt_kits)
# PySide tends to include *all* qt modules, resulting in a 300MB or so folder,
# so we mark them as unwanted, getting us at around 120MB.
qt_excludes = [
    "QtNetwork",
    "QtOpenGL",
    "QtXml",
    "QtTest",
    "QtSql",
    "QtSvg",
    "QtBluetooth",
    "QtDBus",
    "QtDesigner",
    "QtLocation",
    "QtPositioning",
    "QtMultimedia",
    "QtMultimediaWidgets",
    "QtQml",
    "QtQuick",
    "QtSql",
    "QtSvg",
    "QtTest",
    "QtWebKit",
    "QtXml",
    "QtXmlPatterns",
    "QtDeclarative",
    "QtScript",
    "QtScriptTools",
    "QtUiTools",
    "QtQuickWidgets",
    "QtSensors",
    "QtSerialPort",
    "QtWebChannel",
    "QtWebKitWidgets",
    "QtWebSockets",
]
excludes += [f"{qt_api}.{sub}" for sub in qt_excludes]
## Data
# Specify additional "data" that we want to copy over.
# Better to let PyInstaller copy it rather than copying it after the fact.
data1 = {}  # Applied via PyInstaller
data2 = {}  # Manyally copied at the end
data1["_settings"] = "_settings"
# Anything that has .py files should be in data2 on MacOS,
# see https://github.com/pyzo/pyzo/issues/830
if sys.platform.startswith("darwin"):
    data2["../pyzo"] = "source/pyzo"
else:
    data1["../pyzo"] = "source/pyzo"
# Good to first clean up
count = 0
for data_dir in set(data1.keys()) | set(data2.keys()):
    data_dir = os.path.abspath(os.path.join(this_dir, data_dir))
    if os.path.isdir(data_dir):
        for root, dirnames, filenames in os.walk(data_dir):
            for dirname in dirnames:
                if dirname == "__pycache__":
                    shutil.rmtree(os.path.join(root, dirname))
                    count += 1
print(f"removed {count} __pycache__ dirs")
## Create spec
import PyInstaller.__main__
import PyInstaller.utils.cliutils.makespec
entrypoint_pyinstaller = PyInstaller.__main__.run
entrypoint_makespec = PyInstaller.utils.cliutils.makespec.run
# Clear first
if os.path.isdir(dist_dir):
    shutil.rmtree(dist_dir)
os.makedirs(dist_dir)
# Build command
cmd = ["--onedir", "--name", name]
for m in includes:
    cmd.extend(["--hidden-import", m])
for m in excludes:
    cmd.extend(["--exclude-module", m])
for src, dst in data1.items():
    cmd.extend(["--add-data", f"{src}{os.pathsep}{dst}"])
if sys.platform.startswith("win"):
    cmd.append("--windowed")  # not a console app
    cmd.extend(["--icon", icon_file])
elif sys.platform.startswith("darwin"):
    cmd.append("--windowed")  # makes a .app bundle
    cmd.extend(["--icon", icon_file[:-3] + "icns"])
    cmd.extend(["--osx-bundle-identifier", "org.pyzo.app"])
cmd.append(exe_script)
sys.argv[1:] = cmd
entrypoint_makespec()
## Fix spec
specfilename = os.path.join(this_dir, f"{name}.spec")
with open(specfilename, "rb") as f:
    spec = f.read().decode()
if sys.platform.startswith("darwin"):
    i = spec.find("bundle_identifier=")
    assert i > 0
    spec = spec[:i] + f"version='{get_pyzo_version()}',\n             " +
spec[i:]
with open(specfilename, "wb") as f:
    f.write(spec.encode())
## Freeze
```

```
entrypoint_pyinstaller(["--clean", "--distpath", dist_dir, specfilename])
# try:
#     os.remove(specfilename)
# except Exception:
#     pass
## Copy data after freezing
if sys.platform.startswith("darwin"):
    source_dir = os.path.join(dist_dir, "pyzo.app", "Contents", "Resources")
else:
    source_dir = os.path.join(dist_dir, "pyzo")
for dir1, dir2 in data2.items():
    shutil.copytree(
        os.path.abspath(os.path.join(this_dir, dir1)),
        os.path.abspath(os.path.join(source_dir, dir2)),
    )
    print("Copied", dir1, "->", dir2)
# In the GH Action we perform a sign again
```

```
import os
import re
import sys
import zipfile
import tarfile
import platform
import subprocess
this_dir = os.path.abspath(os.path.dirname(__file__)) + "/"
dist_dir = this_dir + "dist/"
with open(os.path.join(this_dir, "..", "pyzo", "__init__.py")) as fh:
    __version__ = re.search(r"__version__ = \"(.*?)\"", fh.read()).group(1)
bitness = "32" if sys.maxsize <= 2**32 else "64"
osname = os.getenv("PYZO_OSNAME", "")
if osname:
    pass
elif sys.platform.startswith("linux"):
    osname = "linux_" + platform.machine()
elif sys.platform.startswith("win"):
    osname = f"win{bitness}"
elif sys.platform.startswith("darwin"):
    osname = "macos_" + platform.machine()
else:
    raise RuntimeError("Unknown platform")
basename = f"pyzo-{__version__}-{osname}"
## Utils
def package_tar_gz():
    print("Packing up into tar.gz ...")
    oridir = os.getcwd()
    os.chdir(dist_dir)
    try:
        tf = tarfile.open(basename + ".tar.gz", "w|gz")
        with tf:
            tf.add("pyzo", arcname="pyzo")
    finally:
        os.chdir(oridir)
def package_zip():
    print("Packing up into zip ...")
    dirname1 = "pyzo.app" if sys.platform.startswith("darwin") else "pyzo"
    dirname2 = dirname1
    zf = zipfile.ZipFile(
        os.path.join(dist_dir, basename + ".zip"), "w",
compression=zipfile.ZIP_DEFLATED
    )
    with zf:
        for root, dirs, files in os.walk(os.path.join(dist_dir, dirname1)):
            for fname in files:
                filename1 = os.path.join(root, fname)
                filename2 = os.path.relpath(filename1, os.path.join(dist_dir, 
dirname1))
                filename2 = os.path.join(dirname2, filename2)
                zf.write(filename1, filename2)
def package_inno_installer():
    print("Packing up into exe installer (via Inno Setup) ...")
    exes = [
        r"c:\Program Files (x86)\Inno Setup 5\ISCC.exe",
        r"c:\Program Files (x86)\Inno Setup 6\ISCC.exe",
    ]
    for exe in exes:
        if os.path.isfile(exe):
            break
    else:
        raise RuntimeError("Could not find Inno Setup exe")
    # Set inno file
    innoFile1 = os.path.join(this_dir, "installerBuilderScript.iss")
    innoFile2 = os.path.join(this_dir, "installerBuilderScript2.iss")
    text = open(innoFile1, "rb").read().decode()
    text = text.replace("X.Y.Z", __version__).replace("64", bitness)
    if bitness == "32":
        text = text.replace("ArchitecturesInstallIn64BitMode = x64", "")
    with open(innoFile2, "wb") as f:
        f.write(text.encode())
    try:
        subprocess.check_call([exe, "/Qp", innoFile2], cwd=dist_dir)
    finally:
        os.remove(innoFile2)
def package_dmg():
    print("Packing up into DMG ...")
    app_dir = "pyzo.app"
    dmg_file = basename + ".dmg"
    cmd = ["hdiutil", "create"]
    cmd.extend(["-srcfolder", app_dir])
    cmd.extend(["-volname", "pyzo"])
    cmd.extend(["-format", "UDZO"])
    cmd.extend(["-fs", "HFSX"])
    # cmd.extend(["-uid", "99"])  # who ever is mounting
    # cmd.extend(["-gid", "99"])  # who ever is mounting
    cmd.extend(["-mode", "555"])  # readonly
    cmd.append("-noscrub")
    cmd.append(dmg_file)
    subprocess.check_call(cmd, cwd=dist_dir)
## Build
if sys.platform.startswith("linux"):
    package_zip()
    package_tar_gz()
if sys.platform.startswith("win"):
    package_zip()
    if bitness == "64":
        # Note: for some reason the 32bit installer is broken. Ah well, the zip works.
        package_inno_installer()
if sys.platform.startswith("darwin"):
    package_zip()
    package_dmg()
```

```
import os
import sys
import subprocess
this_dir = os.path.abspath(os.path.dirname(__file__))
dist_dir = os.path.join(this_dir, "dist")
# Get what executable to run
if sys.platform.startswith("win"):
    exe = os.path.join(dist_dir, "pyzo", "pyzo.exe")
elif sys.platform.startswith("darwin"):
    exe = os.path.join(dist_dir, "pyzo.app", "Contents", "MacOS", "pyzo")
else:
    exe = os.path.join(dist_dir, "pyzo", "pyzo")
# Prepare log file
logfilename = os.path.abspath(os.path.join(__file__, "..", "..", "log.txt"))
with open(logfilename, "wt") as f:
    f.write("")
# Run Pyzo
os.environ["PYZO_LOG"] = logfilename
subprocess.run([exe, "--test"], cwd=this_dir)
```

```
import os
# Automatically scale along on HDPI displays. See issue #531 and e.g.
# https://wiki.archlinux.org/index.php/HiDPI#Qt_5
if "QT_AUTO_SCREEN_SCALE_FACTOR" not in os.environ:
    os.environ["QT_AUTO_SCREEN_SCALE_FACTOR"] = "1"
# Fix Qt now showing a window on MacOS Big Sur
os.environ["QT_MAC_WANTS_LAYER"] = "1"
```

```
""" This is a bit awkward, but yoton is a package that is designed to
work from Python 2.4 to Python 3.x. As such, it does not have relative
imports and must be imported as an absolute package. That is what this
module does...
"""
import os
import sys
# Allow importing yoton
sys.path.insert(0, os.path.dirname(__file__))
# Import yoton
import yoton  # noqa
# Reset
sys.path.pop(0)
```

```
import os
import sys
import ctypes
import locale
import traceback
import pyzo
# Import this module that applies some tweaks that need to be applied
# before import qt. This is a separate module, so that the frozen app
# can import before checking the qt import.
from . import pre_qt_import  # noqa: F401
# Import yoton as an absolute package
from pyzo import yotonloader  # noqa
from pyzo.util import paths
# If there already is an instance of Pyzo, and the user is trying an
# Pyzo command, we should send the command to the other process and quit.
# We do this here, were we have not yet loaded Qt, so we are very light.
from pyzo.core import commandline
if commandline.is_our_server_running():
    print("Started our command server")
else:
    # Handle command line args now
    res = commandline.handle_cmd_args()
    if res:
        print(res)
        sys.exit()
    else:
        # No args, proceed with starting up
        print("Our command server is *not* running")
from pyzo.util import zon as ssdf  # zon is ssdf-light
from pyzo.qt import QtCore, QtGui, QtWidgets
# Enable high-res displays
try:
    ctypes.windll.shcore.SetProcessDpiAwareness(1)
    ctypes.windll.shcore.SetProcessDpiAwareness(2)
except Exception:
    pass  # fail on non-windows
try:
    QtWidgets.QApplication.setAttribute(QtCore.Qt.AA_EnableHighDpiScaling, True)
    QtCore.QCoreApplication.setAttribute(QtCore.Qt.AA_UseHighDpiPixmaps, True)
except Exception:
    pass  # fail on older Qt's
# Import language/translation tools
from pyzo.util._locale import translate, setLanguage  # noqa
```

```
pyzo.translate = translate
pyzo.setLanguage = setLanguage
# Set environ to let kernel know some stats about us
os.environ["PYZO_PREFIX"] = sys.prefix
_is_pyqt4 = hasattr(QtCore, "PYQT_VERSION_STR")
os.environ["PYZO_QTLIB"] = "PyQt4" if _is_pyqt4 else "PySide"
class MyApp(QtWidgets.QApplication):
    """So we an open .py files on OSX.
    OSX is smart enough to call this on the existing process.
    """
    def event(self, event):
        if isinstance(event, QtGui.QFileOpenEvent):
            fname = str(event.file())
            if fname and fname != "pyzo":
                sys.argv[1:] = []
                sys.argv.append(fname)
                res = commandline.handle_cmd_args()
                if not commandline.is_our_server_running():
                    print(res)
                    sys.exit()
        return QtWidgets.QApplication.event(self, event)
if not sys.platform.startswith("darwin"):
    MyApp = QtWidgets.QApplication  # noqa
## Install excepthook
# In PyQt5 exceptions in Python will cause an abort
# http://pyqt.sourceforge.net/Docs/PyQt5/incompatibilities.html
def pyzo_excepthook(type, value, tb):
    out = "Uncaught Python exception: " + str(value) + "\n"
    out += "".join(traceback.format_list(traceback.extract_tb(tb)))
    out += "\n"
    sys.stderr.write(out)
sys.excepthook = pyzo_excepthook
## Define some functions
# todo: move some stuff out of this module ...
def getResourceDirs():
    """getResourceDirs()
    Get the directories to the resources: (pyzoDir, appDataDir, appConfigDir).
    Also makes sure that the appDataDir has a "tools" directory and
    a style file.
    """
    pyzoDir = os.path.abspath(os.path.dirname(__file__))
    if ".zip" in pyzoDir:
        raise RuntimeError("The Pyzo package cannot be run from a zipfile.")
    # Get where the application data is stored (use old behavior on Mac)
    appDataDir, appConfigDir = paths.appdata_dir("pyzo", roaming=True,
macAsLinux=True)
    # Create tooldir if necessary
    toolDir = os.path.join(appDataDir, "tools")
    os.makedirs(toolDir, exist_ok=True)
    return pyzoDir, appDataDir, appConfigDir
def resetConfig(preserveState=True):
    """resetConfig()
    Deletes the config file to revert to default and prevent Pyzo from storing
    its config on the next shutdown.
    """
    # Get filenames
    configFileName2 = os.path.join(pyzo.appConfigDir, "config.ssdf")
    os.remove(configFileName2)
    pyzo._saveConfigFile = False
    print("Deleted user config file. Restart Pyzo to revert to the default
config.")
def loadThemes():
    """
    Load default and user themes (if exist)
    """
    def loadThemesFromDir(dname, isBuiltin=False):
        if not os.path.isdir(dname):
            return
        for fname in [fname for fname in os.listdir(dname) if
fname.endswith(".theme")]:
            try:
                theme = ssdf.load(os.path.join(dname, fname))
                assert (
                    theme.name.lower() == fname.lower().split(".")[0]
                ), "Theme name does not match filename"
                theme.data = {
                    key.replace("_", "."): val for key, val in
theme.data.items()
                }
                theme["builtin"] = isBuiltin
                pyzo.themes[theme.name.lower()] = theme
                print("Loaded theme %r" % theme.name)
            except Exception as ex:
                print("Warning ! Error while reading %s: %s" % (fname, ex))
    loadThemesFromDir(os.path.join(pyzo.pyzoDir, "resources", "themes"), True)
    loadThemesFromDir(os.path.join(pyzo.appDataDir, "themes"))
```

```
def loadConfig(defaultsOnly=False):
    """loadConfig(defaultsOnly=False)
    Load default and site-wide configuration file(s) and that of the user (if it
exists).
    Any missing fields in the user config are set to the defaults.
    """
    # Function to insert names from one config in another
    def replaceFields(base, new):
        for key in new:
            if key in base and isinstance(base[key], ssdf.Struct):
                replaceFields(base[key], new[key])
            else:
                base[key] = new[key]
    config = pyzo.config
    # Reset our pyzo.config structure
    ssdf.clear(config)
    # Load default and inject in the pyzo.config
    fname = os.path.join(pyzo.pyzoDir, "resources", "defaultConfig.ssdf")
    defaultConfig = ssdf.load(fname)
    replaceFields(config, defaultConfig)
    # Platform specific keybinding: on Mac, Ctrl+Tab (actually Cmd+Tab) is a
system shortcut
    if sys.platform == "darwin":
        config.shortcuts2.view__select_previous_file = "Alt+Tab,"
    # Load site-wide config if it exists and inject in pyzo.config
    fname = os.path.join(pyzo.pyzoDir, "resources", "siteConfig.ssdf")
    if os.path.isfile(fname):
        try:
            siteConfig = ssdf.load(fname)
            replaceFields(config, siteConfig)
        except Exception:
            t = "Error while reading config file %r, maybe its corrupt?"
            print(t % fname)
            raise
    # Load user config and inject in pyzo.config
    fname = os.path.join(pyzo.appConfigDir, "config.ssdf")
    if os.path.isfile(fname):
        try:
            userConfig = ssdf.load(fname)
            replaceFields(config, userConfig)
        except Exception:
            t = "Error while reading config file %r, maybe its corrupt?"
            print(t % fname)
            raise
def saveConfig():
    """saveConfig()
    Save all configureations to file.
    """
    # Let the editorStack save its state
    if pyzo.editors:
        pyzo.editors.saveEditorState()
    # Let the main window save its state
    if pyzo.main:
        pyzo.main.saveWindowState()
    # Store config
    if pyzo._saveConfigFile:
        ssdf.save(os.path.join(pyzo.appConfigDir, "config.ssdf"), pyzo.config)
pyzo.getResourceDirs = getResourceDirs
pyzo.resetConfig = resetConfig
pyzo.loadThemes = loadThemes
pyzo.saveConfig = saveConfig
def start():
    """Run Pyzo."""
    # Do some imports
    import pyzo
    from pyzo.core import pyzoLogging  # noqa - to start logging asap
    from pyzo.core.main import MainWindow
    # Apply users' preferences w.r.t. date representation etc
    # this is required for e.g. strftime("%c")
    # Just using '' does not seem to work on OSX. Thus
    # this odd loop.
    # locale.setlocale(locale.LC_ALL, "")
    for x in ("", "C", "en_US", "en_US.utf8", "en_US.UTF-8"):
        try:
            locale.setlocale(locale.LC_ALL, x)
            break
        except Exception:
            pass
    # # Set to be aware of the systems native colors, fonts, etc.
    # QtWidgets.QApplication.setDesktopSettingsAware(True)
    # Instantiate the application
    QtWidgets.qApp = MyApp(sys.argv)  # QtWidgets.QApplication([])
    # Choose language, get locale
    appLocale = setLanguage(pyzo.config.settings.language)
    # Create main window, using the selected locale
    MainWindow(None, appLocale)
    # In test mode, we close after 5 seconds
    # We also write "Closed" to the log (if a filename is provided) which we use
    # in our tests to determine that Pyzo did a successful run.
    if "--test" in sys.argv:
        close_signal = lambda: print("Stopping")
        if os.getenv("PYZO_LOG", ""):
            close_signal = lambda: open(os.getenv("PYZO_LOG"),
"at").write("Stopping\n")
        pyzo.test_close_timer = t = QtCore.QTimer()
        t.setInterval(5000)
        t.setSingleShot(True)
        t.timeout.connect(lambda: [close_signal(), pyzo.main.close()])
        t.start()
    # Enter the main loop
    if hasattr(QtWidgets.qApp, "exec"):
        QtWidgets.qApp.exec()
    else:
        QtWidgets.qApp.exec_()
## Init
# List of names that are later overriden (in main.py)
pyzo.editors = None  # The editor stack instance
pyzo.shells = None  # The shell stack instance
pyzo.main = None  # The mainwindow
pyzo.icon = None  # The icon
pyzo.parser = None  # The source parser
pyzo.status = None  # The statusbar (or None)
# Get directories of interest
pyzo.pyzoDir, pyzo.appDataDir, pyzo.appConfigDir = getResourceDirs()
# Whether the config file should be saved
pyzo._saveConfigFile = True
# Create ssdf in module namespace, and fill it
pyzo.config = ssdf.new()
loadConfig()
try:
    # uses the fact that float("") raises ValueError to be NOP when
qtscalefactor setting is not set
    os.environ["QT_SCREEN_SCALE_FACTORS"] = str(
        float(pyzo.config.settings.qtscalefactor)
    )
except Exception:
    pass
# Create style dict and fill it
pyzo.themes = {}
```

```
loadThemes()
# Init default style name (set in main.restorePyzoState())
pyzo.defaultQtStyleName = ""
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
"""
Pyzo is a cross-platform Python IDE focused on
interactivity and introspection, which makes it very suitable for
scientific computing. Its practical design is aimed at simplicity and
efficiency.
Pyzo is written in Python 3 and Qt. Binaries are available for Windows,
Linux, and Mac. For questions, there is a discussion group.
**Two components + tools**
Pyzo consists of two main components, the editor and the shell, and uses
a set of pluggable tools to help the programmer in various ways. Some
example tools are source structure, project manager, interactive help,
and workspace.
**Some key features**
* Powerful *introspection* (autocompletion, calltips, interactive help)
* Allows various ways to *run code interactively* or to run a file as a script.
* The shells runs in a *subprocess* and can therefore be interrupted or killed.
* *Multiple shells* can be used at the same time, and can be of different
  Python versions (from v2.4 to 3.x, including pypy)
* Support for using several *GUI toolkits* interactively:
  asyncio, PySide, PySide2, PyQt4, PyQt5, wx, fltk, GTK, Tk, Tornado.
* Run IPython shell or native shell.
* *Full Unicode support* in both editor and shell.
* Various handy *tools*, plus the ability to make your own.
* Matlab-style *cell notation* to mark code sections (by starting a line
  with '##').
"""
# Set version number
__version__ = "4.12.8"
import sys
# Check Python version
if sys.version_info < (3, 6):
    raise RuntimeError("Pyzo requires Python 3.6+ to run.")
def start():
    """Start Pyzo."""
    from ._start import start
    start()
```

```
#!/usr/bin/env python3
"""
Pyzo __main__ module
This module enables starting Pyzo via either "python3 -m pyzo" or
"python3 path/to/pyzo".
"""
import os
import sys
import time
import platform
# Very probably run as a script, either the package or the __main__
# directly. Add parent directory to sys.path and try again.
this_dir = os.path.abspath(os.path.dirname(__file__))
sys.path.insert(0, os.path.split(this_dir)[0])
import pyzo
TESTING = "--test" in sys.argv
def write(*msg):
    print(*msg)
    if os.getenv("PYZO_LOG", ""):
        with open(os.getenv("PYZO_LOG"), "at") as f:
            f.write(" ".join(msg) + "\n")
def main():
    if TESTING:
        dt = time.strftime("%Y-%m-%d %H:%M:%S", time.gmtime())
        write(f"Testing Pyzo source ({dt} UTC)")
        write(platform.platform())
        write(sys.version)
    write(f"Pyzo {pyzo.__version__}")
    pyzo.start()
    write("Stopped")  # may be written to log twice because Pyzo defers stdout
if __name__ == "__main__":
    main()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
"""
The base code editor class.
"""
"""
WRITING EXTENSIONS FOR THE CODE EDITOR
The Code Editor extension mechanism works solely based on inheritance.
Extensions can override event handlers (e.g. paintEvent, keyPressEvent). Their
default behaviour should be to call their super() event handler. This way,
events propagate through the extensions following Python's method resolution
order (http://www.python.org/download/releases/2.3/mro/).
A 'fancy' code editor with extensions is created like:
class FancyEditor( Extension1, Extension2, ... CodeEditorBase):
    pass
The order of the extensions does usually matter! If multiple Extensions process
the same key press, the first one has the first chance to consume it.
OVERRIDING __init__
An extensions' __init__ method (if required) should look like this:
class Extension:
    def __init__(self, *args, extensionParam1 = 1, extensionParam2 = 3, **kwds):
        super().__init__(*args, **kwds)
        some_extension_init_stuff()
Note the following points:
 - All parameters have default values
 - The use of *args passes all non-named arguments to its super(), which
   will therefore end up at the QPlainTextEdit constructor. As a consequence,
   the parameters of the exentsion can only be specified as named arguments
 - The use of **kwds ensures that parametes that are not defined by this
   extension, are passed to the next extension(s) in line.
 - The call to super().__init__ is the first thing to do, this ensures that at
   least the CodeEditorBase and QPlainTextEdit, of which the CodeEditorBase is
   derived, are initialized when the initialization of the extension is done
OVERRIDING keyPressEvent
When overriding keyPressEvent, the extension has several options when an event
arrives:
 - Ignore the event
     In this case, call super().keyPressEvent(event) for other extensions or the
     CodeEditorBase to process the event
 - Consume the event
```

```
     In order to prevent other next extensions or the CodeEditorBase to react
     on the event, return without calling the super().keyPressEvent
 - Do something based on the event, and do not let the event propagate
     In this case, do whatever action is defined by the extension, and do not
     call the super().keyPressEvent
 - Do something based on the event, and let the event propagate
     In this case, do whatever action is defined by the extension, and do call
     the super().keyEvent
In any case, the keyPressEvent should not return a value (i.e., return None).
Furthermore, an extension may also want to perform some action *after* the
event has been processed by the next extensions and the CodeEditorBase. In this
case, perform that action after calling super().keyPressEvent
OVERRIDING paintEvent
Then overriding the paintEvent, the extension may want to paint either behind or
in front of the CodeEditorBase text. In order to paint behind the text, first
perform the painting, and then call super().paintEvent. In order to paint in
front of the text, first call super().paintEvent, then perform the painting.
As a result, the total paint order is as follows for the example of the
FancyEditor defined above:
- First the extensions that draw behind the text (i.e. paint before calling
  super().paintEvent, in the order Extension1, Extension2, ...
- then the CodeEditorBase, with the text
- then the extensions that draw in front of the text (i.e. call
  super().paintEvent before painting), in the order ..., Extension2, Extension1
OVERRIDING OTHER EVENT HANDLERS
When overriding other event handlers, be sure to call the super()'s event
handler; either before or after your own actions, as appropriate
OTHER ISSUES
In order to avoid namespace clashes among the extensions, take the following
into account:
 - Private members should start with __ to make ensure no clashes will occur
 - Public members / methods should have names that clearly indicate which
   extension they belong to (e.g. not cancel but autocompleteCancel)
 - Arguments of the __init__ method should also have clearly destictive names
"""
from .qt import QtGui, QtCore, QtWidgets
Qt = QtCore.Qt
from .misc import DEFAULT_OPTION_NAME, DEFAULT_OPTION_NONE, ce_option
from .misc import callLater, ustr
from .manager import Manager
from .highlighter import Highlighter
from .style import StyleElementDescription
class CodeEditorBase(QtWidgets.QPlainTextEdit):
    """The base code editor class. Implements some basic features required
    by the extensions.
    """
    # Style element for default text and editor background
    _styleElements = [
        (
            "Editor.text",
            "The style of the default text. "
            + "One can set the background color here.",
            "fore:#000,back:#fff",
        )
    ]
    # Signal emitted after style has changed
    styleChanged = QtCore.Signal()
    # Signal emitted after font (or font size) has changed
    fontChanged = QtCore.Signal()
    # Signal to indicate a change in breakpoints. Only emitted if the
    # appropriate extension is in use
    breakPointsChanged = QtCore.Signal(object)
    def __init__(self, *args, **kwds):
        super(CodeEditorBase, self).__init__(*args)
        # Set font (always monospace)
        self.__zoom = 0
        self.setFont()
        # Create highlighter class
        self.__highlighter = Highlighter(self, self.document())
        # Set some document options
        # Setting this option breaks the showWhitespace and showLineEndings
options in PySyde6.4
        option = QtGui.QTextOption()  # self.document().defaultTextOption()
        # option.setFlags(
        #     option.flags()
        #     | option.IncludeTrailingSpaces
        #     | option.AddSpaceForLineAndParagraphSeparators
        # )
        self.document().setDefaultTextOption(option)
        # When the cursor position changes, invoke an update, so that
        # the hihghlighting etc will work
        self.cursorPositionChanged.connect(self.viewport().update)
        # Init styles to default values
        self.__style = {}
        for element in self.getStyleElementDescriptions():
            self.__style[element.key] = element.defaultFormat
        # Connext style update
        self.styleChanged.connect(self.__afterSetStyle)
        self.__styleChangedPending = False
        # Init margins
        self._leftmargins = []
        # Init options now.
        # NOTE TO PEOPLE DEVELOPING EXTENSIONS:
        # If an extension has an __init__ in which it first calls the
        # super().__init__, this __initOptions() function will be called,
        # while the extension's init is not yet finished.
        self.__initOptions(kwds)
        # Define colors from Solarized theme
        base03 = "#002b36"
        base02 = "#073642"
        base01 = "#586e75"
        base00 = "#657b83"
        base0 = "#839496"
        base1 = "#93a1a1"
        base2 = "#eee8d5"
        base3 = "#fdf6e3"
        yellow = "#b58900"
        orange = "#cb4b16"
        red = "#dc322f"  # noqa
        magenta = "#d33682"
        violet = "#6c71c4"
        blue = "#268bd2"
        cyan = "#2aa198"
        green = "#859900"  # noqa
        if True:  # Light vs dark
            # back1, back2, back3 = base3, base2, base1 # real solarised
            back1, back2, back3 = "#fff", base2, base1  # crispier
            fore1, fore2, fore3, fore4 = base00, base01, base02, base03
        else:
            back1, back2, back3 = base03, base02, base01
            fore1, fore2, fore3, fore4 = base0, base1, base2, base3  # noqa
        # Define style using "Solarized" colors
        S = {}
        S["Editor.text"] = "back:%s, fore:%s" % (back1, fore1)
        S["Syntax.identifier"] = "fore:%s, bold:no, italic:no, underline:no" %
fore1
        S["Syntax.nonidentifier"] = "fore:%s, bold:no, italic:no, underline:no"
% fore2
        S["Syntax.keyword"] = "fore:%s, bold:yes, italic:no, underline:no" %
```

```
fore2
        S["Syntax.builtins"] = "fore:%s, bold:no, italic:no, underline:no" %
fore1
        S["Syntax.instance"] = "fore:%s, bold:no, italic:no, underline:no" %
fore1
        S["Syntax.functionname"] = "fore:%s, bold:yes, italic:no, underline:no"
% fore3
        S["Syntax.classname"] = "fore:%s, bold:yes, italic:no, underline:no" %
orange
        S["Syntax.string"] = "fore:%s, bold:no, italic:no, underline:no" %
violet
        S["Syntax.unterminatedstring"] = (
            "fore:%s, bold:no, italic:no, underline:dotted" % violet
        )
        S["Syntax.python.multilinestring"] = (
            "fore:%s, bold:no, italic:no, underline:no" % blue
        )
        S["Syntax.number"] = "fore:%s, bold:no, italic:no, underline:no" % cyan
        S["Syntax.comment"] = "fore:%s, bold:no, italic:no, underline:no" %
yellow
        S["Syntax.todocomment"] = "fore:%s, bold:no, italic:yes, underline:no" %
magenta
        S["Syntax.python.cellcomment"] = (
            "fore:%s, bold:yes, italic:no, underline:full" % yellow
        )
        S["Editor.Long line indicator"] = "linestyle:solid, fore:%s" % back2
        S["Editor.Highlight current line"] = "back:%s" % back2
        S["Editor.Indentation guides"] = "linestyle:solid, fore:%s" % back2
        S["Editor.Line numbers"] = "back:%s, fore:%s" % (back2, back3)
        # Apply a good default style
        self.setStyle(S)
    # see https://bugreports.qt.io/browse/QTBUG-
57552?focusedCommentId=469402&page=com.atlassian.jira.plugin.system.issuetabpane
ls%3Acomment-tabpanel#comment-469402
    # and
https://code.qt.io/cgit/qt/qtbase.git/tree/src/gui/text/qtextdocument.cpp#n1183
    # and https://doc.qt.io/qt-5/qchar.html
    _plainTextTrans = str.maketrans(
        {"\u2029": "\n", "\u2028": "\n", "\ufdd0": "\n", "\ufdd1": "\n"}
    )
    def toPlainText(self):
        cursor = self.textCursor()
        cursor.select(QtGui.QTextCursor.Document)
        return cursor.selectedText().translate(self._plainTextTrans)
    def _setHighlighter(self, highlighterClass):
        self.__highlighter = highlighterClass(self, self.document())
    ## Options
    def __getOptionSetters(self):
        """Get a dict that maps (lowercase) option names to the setter
        methods.
        """
        # Get all names that can be options
        allNames = set(dir(self))
        nativeNames = set(dir(QtWidgets.QPlainTextEdit))
        names = allNames.difference(nativeNames)
        # Init dict of setter members
        setters = {}
        for name in names:
            # Get name without set
            if name.lower().startswith("set"):
                name = name[3:]
            # Get setter and getter name
            name_set = "set" + name[0].upper() + name[1:]
            name_get = name[0].lower() + name[1:]
            # Check if both present
            if not (name_set in names and name_get in names):
                continue
            # Get members
            member_set = getattr(self, name_set)
            member_get = getattr(self, name_get)
            # Check if option decorator was used and get default value
            for member in [member_set, member_get]:
                if hasattr(member, DEFAULT_OPTION_NAME):
                    defaultValue = member.__dict__[DEFAULT_OPTION_NAME]
                    break
            else:
                continue
            # Set default on both
            member_set.__dict__[DEFAULT_OPTION_NAME] = defaultValue
            member_get.__dict__[DEFAULT_OPTION_NAME] = defaultValue
            # Add to list
            setters[name.lower()] = member_set
        # Done
        return setters
    def __setOptions(self, setters, options):
        """Sets the options, given the list-of-tuples methods and an
        options dict.
        """
        # List of invalid keys
        invalidKeys = []
        # Set options
        for key1 in options:
            key2 = key1.lower()
            # Allow using the setter name
            if key2.startswith("set"):
                key2 = key2[3:]
            # Check if exists. If so, call!
            if key2 in setters:
                fun = setters[key2]
                val = options[key1]
                fun(val)
            else:
                invalidKeys.append(key1)
        # Check if invalid keys were given
        if invalidKeys:
            print("Warning, invalid options given: " + ", ".join(invalidKeys))
    def __initOptions(self, options=None):
        """Init the options with their default values.
        Also applies the docstrings of one to the other.
        """
        # Make options an empty dict if not given
        if not options:
            options = {}
        # Get setters
        setters = self.__getOptionSetters()
        # Set default value
        for member_set in setters.values():
            defaultVal = member_set.__dict__[DEFAULT_OPTION_NAME]
            if defaultVal != DEFAULT_OPTION_NONE:
                try:
                    member_set(defaultVal)
                except Exception:
                    print("Error initing option ", member_set.__name__)
        # Also set using given opions?
        if options:
            self.__setOptions(setters, options)
    def setOptions(self, options=None, **kwargs):
        """setOptions(options=None, **kwargs)
        Set the code editor options (e.g. highlightCurrentLine) using
        a dict-like object, or using keyword arguments (options given
        in the latter overrule opions in the first).
        The keys in the dict are case insensitive and one can use the
        option's setter or getter name.
        """
        # Process options
        if options:
            D = {}
            for key in options:
                D[key] = options[key]
            D.update(kwargs)
        else:
            D = kwargs
        # Get setters
        setters = self.__getOptionSetters()
        # Go
        self.__setOptions(setters, D)
    ## Font
    def setFont(self, font=None):
        """setFont(font=None)
        Set the font for the editor. Should be a monospace font. If not,
        Qt will select the best matching monospace font.
        """
        defaultFont = Manager.defaultFont()
        # Get font object
        if font is None:
            font = defaultFont
        elif isinstance(font, QtGui.QFont):
            pass
        elif isinstance(font, str):
            font = QtGui.QFont(font)
        else:
            raise ValueError("setFont accepts None, QFont or string.")
        # Hint Qt that it should be monospace
        font.setStyleHint(font.TypeWriter, font.PreferDefault)
        # Get family, fall back to default if qt could not produce monospace
        fontInfo = QtGui.QFontInfo(font)
        if fontInfo.fixedPitch():
            family = fontInfo.family()
        else:
            family = defaultFont.family()
        # Get size: default size + zoom
        size = defaultFont.pointSize() + self.__zoom
        # Create font instance
        font = QtGui.QFont(family, size)
        # Set, emit and return
        QtWidgets.QPlainTextEdit.setFont(self, font)
        self.fontChanged.emit()
        return font
    def setZoom(self, zoom):
        """setZoom(zoom)
        Set the zooming of the document. The font size is always the default
        font size + the zoom factor.
        The final zoom is returned, this may not be the same as the given
        zoom factor if the given factor is too small.
        """
        # Set zoom (limit such that final pointSize >= 1)
        size = Manager.defaultFont().pointSize()
        self.__zoom = int(max(1 - size, zoom))
        # Set font
        self.setFont(self.fontInfo().family())
        # Return zoom
        return self.__zoom
    ## Syntax / styling
    @classmethod
    def getStyleElementDescriptions(cls):
        """getStyleElementDescriptions()
        This classmethod returns a list of the StyleElementDescription
        instances used by this class. This includes the descriptions for
        the syntax highlighting of all parsers.
        """
        # Collect members by walking the class bases
        elements = []
        def collectElements(cls, iter=1):
            # Valid class?
            if cls is object or cls is QtWidgets.QPlainTextEdit:
                return
            # Check members
            if hasattr(cls, "_styleElements"):
                for element in cls._styleElements:
                    elements.append(element)
            # Recurse
            for c in cls.__bases__:
                collectElements(c, iter + 1)
        collectElements(cls)
        # Make style element descriptions
        # (Use a dict to ensure there are no duplicate keys)
        elements2 = {}
        for element in elements:
            # Check
            if isinstance(element, StyleElementDescription):
                pass
            elif isinstance(element, tuple):
                element = StyleElementDescription(*element)
            else:
                print("Warning: invalid element: " + repr(element))
            # Store using the name as a key to prevent duplicates
            elements2[element.key] = element
        # Done
        return list(elements2.values())
    def getStyleElementFormat(self, name):
        """getStyleElementFormat(name)
        Get the style format for the style element corresponding with
        the given name. The name is case insensitive and invariant to
        the use of spaces.
        """
        key = name.replace(" ", "").lower()
        try:
            return self.__style[key]
        except KeyError:
            raise KeyError('Not a known style element name: "%s".' % name)
    def setStyle(self, style=None, **kwargs):
        """setStyle(style=None, **kwargs)
        Updates the formatting per style element.
        The style consists of a dictionary that maps style names to
        style formats. The style names are case insensitive and invariant
        to the use of spaces.
        For convenience, keyword arguments may also be used. In this case,
        underscores are interpreted as dots.
        This function can also be called without arguments to force the
        editor to restyle (and rehighlight) itself.
        Use getStyleElementDescriptions() to get information about the
        available styles and their default values.
        Examples
        --------
        # To make the classname in underline, but keep the color and boldness:
        setStyle(syntax_classname='underline')
        # To set all values for function names:
        setStyle(syntax_functionname='#883,bold:no,italic:no')
        # To set line number and indent guides colors
        setStyle({  'editor.LineNumbers':'fore:#000,back:#777',
                    'editor.indentationGuides':'#f88' })
        """
        # Combine user input
        D = {}
        if style:
            for key in style:
                D[key] = style[key]
        if True:
            for key in kwargs:
                key2 = key.replace("_", ".")
                D[key2] = kwargs[key]
        # List of given invalid style element names
        invalidKeys = []
        # Set style elements
        for key in D:
            normKey = key.replace(" ", "").lower()
            if normKey in self.__style:
                # self.__style[normKey] = StyleFormat(D[key])
                self.__style[normKey].update(D[key])
            else:
                invalidKeys.append(key)
        # Give warning for invalid keys
        if invalidKeys:
            print("Warning, invalid style names given: " +
",".join(invalidKeys))
        # Notify that style changed, adopt a lazy approach to make loading
        # quicker.
        if self.isVisible():
            callLater(self.styleChanged.emit)
            self.__styleChangedPending = False
        else:
            self.__styleChangedPending = True
    def showEvent(self, event):
        super(CodeEditorBase, self).showEvent(event)
        # Does the style need updating?
        if self.__styleChangedPending:
            callLater(self.styleChanged.emit)
            self.__styleChangedPending = False
    def __afterSetStyle(self):
        """_afterSetStyle()
        Method to call after the style has been set.
        """
        # Set text style using editor style sheet
        format = self.getStyleElementFormat("editor.text")
        ss = "QPlainTextEdit{ color:%s; background-color:%s; }" % (
            format["fore"],
            format["back"],
        )
        self.setStyleSheet(ss)
        # Make sure the style is applied
        self.viewport().update()
        # Re-highlight
        callLater(self.__highlighter.rehighlight)
    ## Some basic options
    @ce_option(4)
    def indentWidth(self):
        """Get the width of a tab character, and also the amount of spaces
        to use for indentation when indentUsingSpaces() is True.
        """
        return self.__indentWidth
    def setIndentWidth(self, value):
        value = int(value)
        if value <= 0:
            raise ValueError("indentWidth must be >0")
        self.__indentWidth = value
        self.setTabStopWidth(self.fontMetrics().width("i" * self.__indentWidth))
    @ce_option(False)
    def indentUsingSpaces(self):
        """Get whether to use spaces (if True) or tabs (if False) to indent
        when the tab key is pressed
        """
        return self.__indentUsingSpaces
    def setIndentUsingSpaces(self, value):
        self.__indentUsingSpaces = bool(value)
        self.__highlighter.rehighlight()
    ## Misc
    def gotoLine(self, lineNumber):
        """gotoLine(lineNumber)
        Move the cursor to the block given by the line number
        (first line is number 1) and show that line.
        """
        return self.gotoBlock(lineNumber - 1)
    def gotoBlock(self, blockNumber):
        """gotoBlock(blockNumber)
        Move the cursor to the block given by the block number
        (first block is number 0) and show that line.
        """
        # Two implementatios. I know that the latter works, so lets
        # just use that.
        cursor = self.textCursor()
        # block = self.document().findBlockByNumber( blockNumber )
        # cursor.setPosition(block.position())
        cursor.movePosition(cursor.Start)  # move to begin of the document
        cursor.movePosition(cursor.NextBlock, n=blockNumber)  # n blocks down
        try:
            self.setTextCursor(cursor)
        except Exception:
            pass  # File is smaller then the caller thought
        # TODO make this user configurable (setting relativeMargin to anything
above
        # 0.5 will cause cursor to center on each move)
        relativeMargin = 0.2  # 20% margin on both sides of the window
        margin = self.height() * relativeMargin
        cursorRect = self.cursorRect(cursor)
        if cursorRect.top() < margin or cursorRect.bottom() + margin >
self.height():
            self.centerCursor()
    def doForSelectedBlocks(self, function):
        """doForSelectedBlocks(function)
        Call the given function(cursor) for all blocks in the current selection
        A block is considered to be in the current selection if a part of it is
in
        the current selection
        The supplied cursor will be located at the beginning of each block. This
        cursor may be modified by the function as required
        """
        # Note: a 'TextCursor' does not represent the actual on-screen cursor,
so
        # movements do not move the on-screen cursor
        # Note 2: when the text is changed, the cursor and selection start/end
        # positions of all cursors are updated accordingly, so the screenCursor
        # stays in place even if characters are inserted at the editCursor
        screenCursor = self.textCursor()  # For maintaining which region is
selected
        editCursor = self.textCursor()  # For inserting the comment marks
        # Use beginEditBlock / endEditBlock to make this one undo/redo operation
        editCursor.beginEditBlock()
        try:
            editCursor.setPosition(screenCursor.selectionStart())
            editCursor.movePosition(editCursor.StartOfBlock)
            # < :if selection end is at beginning of the block, don't include
that
            # one, except when the selectionStart is same as selectionEnd
            while (
                editCursor.position() < screenCursor.selectionEnd()
                or editCursor.position() <= screenCursor.selectionStart()
            ):
                # Create a copy of the editCursor and call the user-supplied
function
                editCursorCopy = QtGui.QTextCursor(editCursor)
                function(editCursorCopy)
                # Move to the next block
                if not editCursor.block().next().isValid():
                    break  # We reached the end of the document
                editCursor.movePosition(editCursor.NextBlock)
        finally:
            editCursor.endEditBlock()
    def doForVisibleBlocks(self, function):
        """doForVisibleBlocks(function)
        Call the given function(cursor) for all blocks that are currently
        visible. This is used by several appearence extensions that
        paint per block.
        The supplied cursor will be located at the beginning of each block. This
        cursor may be modified by the function as required
        """
        # Start cursor at top line.
        cursor = self.cursorForPosition(QtCore.QPoint(0, 0))
        cursor.movePosition(cursor.StartOfBlock)
        if not self.isVisible():
            return
        while True:
            # Call the function with a copy of the cursor
            function(QtGui.QTextCursor(cursor))
            # Go to the next block (or not if we are done)
            y = self.cursorRect(cursor).bottom()
            if y > self.height():
                break  # Reached end of the repaint area
            if not cursor.block().next().isValid():
                break  # Reached end of the text
            cursor.movePosition(cursor.NextBlock)
    def indentBlock(self, cursor, amount=1):
        """indentBlock(cursor, amount=1)
        Indent the block given by cursor.
        The cursor specified is used to do the indentation; it is positioned
        at the beginning of the first non-whitespace position after completion
        May be overridden to customize indentation.
        """
        text = ustr(cursor.block().text())
        leadingWhitespace = text[: len(text) - len(text.lstrip())]
        # Select the leading whitespace
        cursor.movePosition(cursor.StartOfBlock)
        cursor.movePosition(cursor.Right, cursor.KeepAnchor,
len(leadingWhitespace))
        # Compute the new indentation length, expanding any existing tabs
        indent = len(leadingWhitespace.expandtabs(self.indentWidth()))
        if self.indentUsingSpaces():
            # Determine correction, so we can round to multiples of indentation
            correction = indent % self.indentWidth()
            if correction and amount < 0:
                correction = -(self.indentWidth() - correction)  # Flip
            # Add the indentation tabs
            indent += (self.indentWidth() * amount) - correction
            cursor.insertText(" " * max(indent, 0))
        else:
            # Convert indentation to number of tabs, and add one
            indent = (indent // self.indentWidth()) + amount
            cursor.insertText("\t" * max(indent, 0))
    def dedentBlock(self, cursor):
        """dedentBlock(cursor)
        Dedent the block given by cursor.
        Calls indentBlock with amount = -1.
        May be overridden to customize indentation.
        """
        self.indentBlock(cursor, amount=-1)
    def indentSelection(self):
        """indentSelection()
        Called when the current line/selection is to be indented.
        Calls indentLine(cursor) for each line in the selection.
        May be overridden to customize indentation.
        See also doForSelectedBlocks and indentBlock.
        """
        self.doForSelectedBlocks(self.indentBlock)
    def dedentSelection(self):
        """dedentSelection()
        Called when the current line/selection is to be dedented.
        Calls dedentLine(cursor) for each line in the selection.
        May be overridden to customize indentation.
        See also doForSelectedBlocks and dedentBlock.
        """
        self.doForSelectedBlocks(self.dedentBlock)
    def justifyText(self, linewidth=70):
        """justifyText(linewidth=70)"""
        from .textutils import TextReshaper
        # Get cursor
        cursor = self.textCursor()
        # Make selection include whole lines
        pos1, pos2 = cursor.position(), cursor.anchor()
        pos1, pos2 = min(pos1, pos2), max(pos1, pos2)
        cursor.setPosition(pos1, cursor.MoveAnchor)
        cursor.movePosition(cursor.StartOfBlock, cursor.MoveAnchor)
        cursor.setPosition(pos2, cursor.KeepAnchor)
        cursor.movePosition(cursor.EndOfBlock, cursor.KeepAnchor)
        # Use reshaper to create replacement text
        reshaper = TextReshaper(linewidth)
        reshaper.pushText(cursor.selectedText())
        newText = reshaper.popText()
        # Update the selection
        # self.setTextCursor(cursor) for testing
        cursor.insertText(newText)
    def addLeftMargin(self, des, func):
        """Add a margin to the left. Specify a description for the margin,
        and a function to get that margin. For internal use.
        """
        assert des is not None
        self._leftmargins.append((des, func))
    def getLeftMargin(self, des=None):
        """Get the left margin, relative to the given description (which
        should be the same as given to addLeftMargin). If des is omitted
        or None, the full left margin is returned.
        """
        margin = 0
        for d, func in self._leftmargins:
            if d == des:
                break
            margin += func()
        return margin
    def updateMargins(self):
        """Force the margins to be recalculated and set the viewport
        accordingly.
        """
        leftmargin = self.getLeftMargin()
        self.setViewportMargins(leftmargin, 0, 0, 0)
    def toggleCase(self):
        """Change selected text to lower or upper case."""
        # Get cursor
        cursor = self.textCursor()
        # position = cursor.position()
        start_pos = cursor.selectionStart()
        end_pos = cursor.selectionEnd()
        # Get selected text
        selection = cursor.selectedText()
        if selection.islower():
            newText = selection.upper()
        elif selection.isupper():
            newText = selection.lower()
        else:
            newText = selection.lower()
        # Update the selection
        cursor.insertText(newText)
        cursor.setPosition(start_pos)
        cursor.setPosition(end_pos, QtGui.QTextCursor.KeepAnchor)
        self.setTextCursor(cursor)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module highlighter
Defines the highlighter class for the base code editor class. It will do
the styling when syntax highlighting is enabled. If it is not, will only
check out indentation.
"""
from .qt import QtGui, QtCore
Qt = QtCore.Qt
from . import parsers
from .misc import ustr
class BlockData(QtGui.QTextBlockUserData):
    """Class to represent the data for a block."""
    def __init__(self):
        QtGui.QTextBlockUserData.__init__(self)
        self.indentation = None
        self.fullUnderlineFormat = None
        self.tokens = []
# The highlighter should be part of the base class, because
# some extensions rely on them (e.g. the indent guuides).
class Highlighter(QtGui.QSyntaxHighlighter):
    def __init__(self, codeEditor, *args):
        QtGui.QSyntaxHighlighter.__init__(self, *args)
        # Store reference to editor
        self._codeEditor = codeEditor
    def getCurrentBlockUserData(self):
        """getCurrentBlockUserData()
        Gets the BlockData object. Creates one if necesary.
        """
        bd = self.currentBlockUserData()
        if not isinstance(bd, BlockData):
            bd = BlockData()
            self.setCurrentBlockUserData(bd)
        return bd
    def highlightBlock(self, line):
        """highlightBlock(line)
        This method is automatically called when a line must be
        re-highlighted.
        If the code editor has an active parser. This method will use
        it to perform syntax highlighting. If not, it will only
        check out the indentation.
        """
        # Make sure this is a Unicode Python string
        line = ustr(line)
        # Get previous state
        previousState = self.previousBlockState()
        # Get parser
        parser = None
        if hasattr(self._codeEditor, "parser"):
            parser = self._codeEditor.parser()
        # Get function to get format
        nameToFormat = self._codeEditor.getStyleElementFormat
        fullLineFormat = None
        tokens = []
        if parser:
            self.setCurrentBlockState(0)
            tokens = list(parser.parseLine(line, previousState))
            for token in tokens:
                # Handle block state
                if isinstance(token, parsers.BlockState):
                    self.setCurrentBlockState(token.state)
                else:
                    # Get format
                    try:
                        styleFormat = nameToFormat(token.name)
                        charFormat = styleFormat.textCharFormat
                    except KeyError:
                        # print(repr(nameToFormat(token.name)))
                        continue
                    # Set format
                    self.setFormat(token.start, token.end - token.start,
charFormat)
                    # Is this a cell?
                    if (fullLineFormat is None) and styleFormat._parts.get(
                        "underline", ""
                    ) == "full":
                        fullLineFormat = styleFormat
        # Get user data
        bd = self.getCurrentBlockUserData()
        # Store token list for future use (e.g. brace matching)
        bd.tokens = tokens
        # Handle underlines
        bd.fullUnderlineFormat = fullLineFormat
```

```
# Get the indentation setting of the editors
indentUsingSpaces = self._codeEditor.indentUsingSpaces()
leadingWhitespace = line[: len(line) - len(line.lstrip())]
if "\t" in leadingWhitespace and " " in leadingWhitespace:
    # Mixed whitespace
    bd.indentation = 0
    format = QtGui.QTextCharFormat()
    format.setUnderlineStyle(QtGui.QTextCharFormat.SpellCheckUnderline)
    format.setUnderlineColor(QtCore.Qt.red)
    format.setToolTip("Mixed tabs and spaces")
    self.setFormat(0, len(leadingWhitespace), format)
elif ("\t" in leadingWhitespace and indentUsingSpaces) or (
    " " in leadingWhitespace and not indentUsingSpaces
):
    # Whitespace differs from document setting
    bd.indentation = 0
    format = QtGui.QTextCharFormat()
    format.setUnderlineStyle(QtGui.QTextCharFormat.SpellCheckUnderline)
    format.setUnderlineColor(QtCore.Qt.blue)
    format.setToolTip("Whitespace differs from document setting")
    self.setFormat(0, len(leadingWhitespace), format)
else:
    # Store info for indentation guides
    # amount of tabs or spaces
    bd.indentation = len(leadingWhitespace)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Codeeditor is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module manager
This module contains a static class that can be used for some
management tasks.
"""
import os, sys
from .qt import QtGui, QtCore, QtWidgets  # noqa
Qt = QtCore.Qt
from . import parsers
class Manager:
    """Manager
    Static class to do some management tasks:
      * It manages the parsers
      * Getting style element descriptions of all parsers
      * Linking file extensions to parsers
      * Font information
    """
    _defaultFontFamily = "dummy_font_family_name"
    # Static dict of all parsers
    _parserInstances = {}
    _fileExtensions = {}
    _shebangKeywords = {}
    ## Parsers
    #     @classmethod
    #     def collectParsersDynamically(cls):
    #         """ insert the function is this module's namespace.
    #         """
    #
    #         # Get the path of this subpackage
    #         path = __file__
    #         path = os.path.dirname( os.path.abspath(path) )
    #
    #         # Determine if we're in a zipfile
    #         i = path.find('.zip')
    #         if i>0:
    #             # get list of files from zipfile
    #             path = path[:i+4]
    #             z = zipfile.ZipFile(path)
    #             files = [os.path.split(i)[-1] for i in z.namelist()
    #                         if 'codeeditor' in i and 'parsers' in i]
    #         else:
    #             # get list of files from file system
    #             files = os.listdir(path)
    #
    #         # Extract all parsers
    #         parserModules = []
    #         for file in files:
    #
    #             # Only python files
    #             if file.endswith('.pyc'):
    #                 if file[:-1] in files:
    #                     continue # Only try import once
    #             elif not file.endswith('.py'):
    #                 continue
    #             # Only syntax files
    #             if '_parser.' not in file:
    #                 continue
    #
    #             # Import module
    #             fullfile = os.path.join(path, file)
    #             modname = os.path.splitext(file)[0]
    #             print('modname', modname)
    #             mod = __import__("codeeditor.parsers."+modname,
fromlist=[modname])
    #             parserModules.append(mod)
    #
    #         print(parserModules)
    @classmethod
    def _collectParsers(cls):
        """_collectParsers()
        Collect all parser classes. This function is called on startup.
        """
        # Prepare (use a set to prevent duplicates)
        foundParsers = set()
        G = parsers.__dict__
        ModuleClass = os.__class__
        # Collect parser classes
        for module_name in G:
            # Check if it is indeed a module, and if it has the right name
            if not isinstance(G[module_name], ModuleClass):
                continue
            if not module_name.endswith("_parser"):
                continue
            # Collect all valid classes from the module
            moduleDict = G[module_name].__dict__
            for name_in_module in moduleDict:
                ob = moduleDict[name_in_module]
                if isinstance(ob, type) and issubclass(ob, parsers.Parser):
                    foundParsers.add(ob)
        # Put in list with the parser names as keys
        parserInstances = {}
        for parserClass in foundParsers:
            name = parserClass.getParserName()
            if name:
                # Try instantiating the parser
                try:
                    parserInstances[name] = parserInstance = parserClass()
                except Exception:
                    # We cannot get the exception object in a Python2/Python3
                    # compatible way
                    print('Could not instantiate parser "%s".' % name)
                    continue
                # Register extensions and shebang keywords for this parser
                for ext in parserInstance.filenameExtensions():
                    cls._fileExtensions[ext] = name
                for skw in parserInstance.shebangKeywords():
                    cls._shebangKeywords[skw] = name
        # Store
        cls._parserInstances = parserInstances
    @classmethod
    def getParserNames(cls):
        """getParserNames()
        Get a list of all available parsers.
        """
        return list(cls._parserInstances.keys())
    @classmethod
    def getParserByName(cls, parserName):
        """getParserByName(parserName)
        Get the parser object corresponding to the given name.
        If no parser is known by the given name, a warning message
        is printed and None is returned.
        """
        if not parserName:
            return parsers.Parser()  # Default dummy parser
        # Case insensitive
        parserName = parserName.lower()
        # Return instantiated parser object.
        if parserName in cls._parserInstances:
            return cls._parserInstances[parserName]
        else:
            print('Warning: no parser known by the name "%s".' % parserName)
            print("I know these: ", cls._parserInstances.keys())
            return parsers.Parser()  # Default dummy parser
    @classmethod
    def getStyleElementDescriptionsForAllParsers(cls):
        """getStyleElementDescriptionsForAllParsers()
        Get all style element descriptions corresponding to
        the tokens of all parsers.
        This function is used by the code editor to register all syntax
        element styles to the code editor class.
        """
        descriptions = {}
        for parser in cls._parserInstances.values():
            for token in parser.getUsedTokens():
                description = token.description
                descriptions[description.key] = description
        return list(descriptions.values())
    ## File extensions
    @classmethod
    def suggestParserfromFilenameExtension(cls, ext):
        """suggestParserfromFilenameExtension(ext)
        Given a filename extension, returns the name of the suggested
        parser corresponding to the language of the file.
        See also registerFilenameExtension()
        """
        # Normalize ext
        ext = "." + ext.lstrip(".").lower()
        # Get parser
        if ext in cls._fileExtensions:
            return cls._fileExtensions[ext]
        else:
            return ""
    @classmethod
    def suggestParserfromText(cls, text):
        """suggestParserfromText(text)
        Given a text, returns the name of the suggested
        parser corresponding to the language of the file.
        See also registerShebangKeyword()
        """
        shebangline = None
        for line in text[:1000].splitlines():
            line = line.strip()
            if line.startswith("#!"):
                shebangline = line
                break
        if shebangline is None:
            return ""
        shebangline = shebangline[2:].split()  # takes care of eventual space
after #!
        if len(shebangline) == 0:
            return ""
        interpreter = os.path.basename(shebangline[0])
        if interpreter == "env" and len(shebangline) > 1:
            interpreter = shebangline[1]
        # Get parser
        if interpreter in cls._shebangKeywords:
            return cls._shebangKeywords[interpreter]
        else:
            return ""
    @classmethod
    def suggestParser(cls, ext, text):
        """suggestParser(ext, text)
        Given a filename extension and text, returns the name of the suggested
        parser corresponding to the language of the file.
        See also registerFilenameExtension() and registerShebangKeyword()
        """
        parser = cls.suggestParserfromText(text)
        if parser == "":
            parser = cls.suggestParserfromFilenameExtension(ext)
        parser = cls.getParserByName(parser).disambiguate(text)
        return parser
    @classmethod
    def registerFilenameExtension(cls, ext, parser):
        """registerFilenameExtension(ext, parser)
        Registers the given filename extension to the given parser.
        The parser can be a Parser instance or its name.
        This function can be used to register extensions to parsers
        that are not registered by default.
        """
        # Normalize ext
        ext = "." + ext.lstrip(".").lower()
        # Check parser
        if isinstance(parser, parsers.Parser):
            parser = parser.name()
        # Register
        cls._fileExtensions[ext] = parser
    @classmethod
    def registerShebangKeyword(cls, shebangKeyword, parser):
        """registerShebangKeyword(shebangKeyword, parser)
        Registers the given shebang keyword (interpreter) to the given parser.
        The parser can be a Parser instance or its name.
        This function can be used to register shebang keywords to parsers
        that are not registered by default.
        """
        # Check parser
        if isinstance(parser, parsers.Parser):
            parser = parser.name()
        # Register
        cls._shebangKeywords[shebangKeyword] = parser
    ## Fonts
    @classmethod
    def fontNames(cls):
        """fontNames()
        Get a list of all monospace fonts available on this system.
        """
        db = QtGui.QFontDatabase()
        return [fn for fn in db.families() if db.isFixedPitch(fn)]
    @classmethod
    def setDefaultFontFamily(cls, name):
        """setDefaultFontFamily(name)
        Set the default (monospace) font family name for this system.
        This should be set only once during startup.
        """
        cls._defaultFontFamily = name
    @classmethod
    def defaultFont(cls):
        """defaultFont()
        Get the default (monospace) font for this system. Returns a QFont
        object.
        """
        # Get font family
        f = QtGui.QFont(cls._defaultFontFamily)
        f.setStyleHint(f.TypeWriter, f.PreferDefault)
        fi = QtGui.QFontInfo(f)
        family = fi.family()
        # Get the font size
        size = 9
        if sys.platform.startswith("darwin"):
            # Account for Qt font size difference
            # http://qt-project.org/forums/viewthread/27201
            # Win/linux use 96 ppi, OS X uses 72 -> 133% ratio
            size = int(size * 1.33333 + 0.4999)
        # Done
        return QtGui.QFont(family, size)
# Init
try:
    Manager._collectParsers()
except Exception as why:
    print("Error collecting parsers")
    print(why)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module misc
Defined ustr (Unicode string) class and the option property decorator.
"""
import sys
from .qt import QtGui, QtCore, QtWidgets  # noqa
# Set Python version and get some names
PYTHON_VERSION = sys.version_info[0]
if PYTHON_VERSION < 3:
    ustr = unicode  # noqa
    bstr = str
    from Queue import Queue, Empty
else:
    ustr = str
    bstr = bytes
    from queue import Queue, Empty
DEFAULT_OPTION_NAME = "_ce_default_value"
DEFAULT_OPTION_NONE = "_+_just some absurd value_+_"
def ce_option(arg1):
    """Decorator for properties of the code editor.
    It should be used on the setter function, with its default value
    as an argument. The default value is then  stored on the function
    object.
    At the end of the initialization, the base codeeditor class will
    check all members and (by using the default-value-attribute as a
    flag) select the ones that are options. These are then set to
    their default values.
    Similarly this information is used by the setOptions method to
    know which members are "options".
    """
    # If the decorator is used without arguments, arg1 is the function
    # being decorated. If arguments are used, arg1 is the argument, and
    # we should return a callable that is then used as a decorator.
    # Create decorator function.
    def decorator_fun(f):
        f.__dict__[DEFAULT_OPTION_NAME] = default
        return f
    # Handle
    default = DEFAULT_OPTION_NONE
    if hasattr(arg1, "__call__"):
        return decorator_fun(arg1)
    else:
        default = arg1
        return decorator_fun
class _CallbackEventHandler(QtCore.QObject):
    """Helper class to provide the callLater function."""
    def __init__(self):
        QtCore.QObject.__init__(self)
        self.queue = Queue()
    def customEvent(self, event):
        while True:
            try:
                callback, args = self.queue.get_nowait()
            except Empty:
                break
            try:
                callback(*args)
            except Exception as why:
                print("callback failed: {}:\n{}".format(callback, why))
    def postEventWithCallback(self, callback, *args):
        self.queue.put((callback, args))
        QtWidgets.qApp.postEvent(self, QtCore.QEvent(QtCore.QEvent.User))
def callLater(callback, *args):
    """callLater(callback, *args)
    Post a callback to be called in the main thread.
    """
    _callbackEventHandler.postEventWithCallback(callback, *args)
# Create callback event handler instance and insert function in Pyzo namespace
_callbackEventHandler = _CallbackEventHandler()
```

```
# This is the one place where codeeditor depends on Pyzo itself
from pyzo.qt import QtCore, QtGui, QtWidgets  # noqa
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Modyule style
Provides basic functionaliy for styling.
Styling is done using a dictionary of StyleFormat instances. Each
such instance reprsents a certain element being styled (e.g. keywords,
line numbers, indentation guides).
All possible style elements are represented using StyleElementDescription
instances. These have a name, description and default format, which
makes it easy to build a UI to allow the user to change the syle.
"""
from .qt import QtGui, QtCore
Qt = QtCore.Qt
class StyleElementDescription:
    """StyleElementDescription(name, defaultFormat, description)
    Describes a style element by its name, default format, and description.
    A style description is a simple placeholder for something
    that can be styled.
    """
    def __init__(self, name, description, defaultFormat):
        self._name = name
        self._description = description
        self._defaultFormat = StyleFormat(defaultFormat)
    def __repr__(self):
        return '<"%s": "%s">' % (self.name, self.defaultFormat)
    @property
    def name(self):
        return self._name
    @property
    def key(self):
        return self._name.replace(" ", "").lower()
    @property
    def description(self):
        return self._description
    @property
    def defaultFormat(self):
        return self._defaultFormat
class StyleFormat:
    """StyleFormat(format='')
    Represents the style format for a specific style element.
    A "style" is a dictionary that maps names (of style elements)
    to StyleFormat instances.
    The given format can be a string or another StyleFormat instance.
    Style formats can be combined using their update() method.
    A style format consists of multiple parts, where each "part" consists
    of a key and a value. The keys can be anything, depending
    on what kind of thing is being styled. The value can be obtained using
    the index operator (e.g. styleFomat['fore'])
    For a few special keys, properties are defined that return the Qt object
    corresponding to the value. These values are also buffered to enable
    fast access. These keys are:
      * fore: (QColor) the foreground color
      * back: (QColor) the background color
      * bold: (bool) whether the text should be bold
      * italic: (bool) whether the text should be in italic
      * underline: (int) whether an underline should be used (and which one)
      * linestyle: (int) what line style to use (e.g. for indent guides)
      * textCharFOrmat: (QTextCharFormat) for the syntax styles
    The format neglects spaces and case. Parts are separated by commas
    or semicolons. If only a key is given it's value is interpreted
    as 'yes'. If only a color is given, its key is interpreted as 'fore'
    and back. Colors should be given using the '#' hex formatting.
    An example format string: 'fore:#334, bold, underline:dotLine'
    By calling str(styleFormatInstance) the string representing of the
    format can be obtained. By iterating over the instance, a series
    of key-value pairs is obtained.
    """
    def __init__(self, format=""):
        self._parts = {}
        self.update(format)
    def _resetProperties(self):
        self._fore = None
        self._back = None
        self._bold = None
        self._italic = None
        self._underline = None
        self._linestyle = None
        self._textCharFormat = None
    def __str__(self):
        """Get a (cleaned up) string representation of this style format."""
        parts = []
        for key in self._parts:
            parts.append("%s:%s" % (key, self._parts[key]))
        return ", ".join(parts)
    def __repr__(self):
        return '<StyleFormat "%s">' % str(self)
    def __getitem__(self, key):
        try:
            return self._parts[key]
        except KeyError:
            raise KeyError("Invalid part key " + key + " for style format.")
    def __iter__(self):
        """Yields a series of tuples (key, val)."""
        parts = []
        for key in self._parts:
            parts.append((key, self._parts[key]))
        return parts.__iter__()
    def update(self, format):
        """update(format)
        Update this style format with the given format.
        """
        # Reset buffered values
        self._resetProperties()
        # Make a string, so we update the format with the given one
        if isinstance(format, StyleFormat):
            format = str(format)
        # Split on ',' and ',', ignore spaces
        styleParts = [p for p in format.replace("=", ":").replace(";",
",").split(",")]
        for stylePart in styleParts:
            # Make sure it consists of identifier and value pair
            # e.g. fore:#xxx, bold:yes, underline:no
            if ":" not in stylePart:
                if stylePart.startswith("#"):
                    stylePart = "foreandback:" + stylePart
                else:
                    stylePart += ":yes"
            # Get key value and strip and make lowecase
            key, _, val = [i.strip().lower() for i in stylePart.partition(":")]
            # Store in parts
            if key == "foreandback":
                self._parts["fore"] = val
                self._parts["back"] = val
            elif key:
                self._parts[key] = val
    ## Properties
    def _getValueSafe(self, key):
        try:
            return self._parts[key]
        except KeyError:
            return "no"
    @property
    def fore(self):
        if self._fore is None:
            self._fore = QtGui.QColor(self._parts["fore"])
        return self._fore
    @property
    def back(self):
        if self._back is None:
            self._back = QtGui.QColor(self._parts["back"])
        return self._back
    @property
    def bold(self):
        if self._bold is None:
            if self._getValueSafe("bold") in ["yes", "true"]:
                self._bold = True
            else:
                self._bold = False
        return self._bold
    @property
    def italic(self):
        if self._italic is None:
            if self._getValueSafe("italic") in ["yes", "true"]:
                self._italic = True
            else:
                self._italic = False
        return self._italic
    @property
    def underline(self):
        if self._underline is None:
            val = self._getValueSafe("underline")
            if val in ["yes", "true"]:
                self._underline = QtGui.QTextCharFormat.SingleUnderline
            elif val in ["dotted", "dots", "dotline"]:
                self._underline = QtGui.QTextCharFormat.DotLine
            elif val in ["wave"]:
                self._underline = QtGui.QTextCharFormat.WaveUnderline
            else:
                self._underline = QtGui.QTextCharFormat.NoUnderline
        return self._underline
    @property
    def linestyle(self):
        if self._linestyle is None:
            val = self._getValueSafe("linestyle")
            if val in ["yes", "true"]:
                self._linestyle = Qt.SolidLine
            elif val in ["dotted", "dot", "dots", "dotline"]:
                self._linestyle = Qt.DotLine
            elif val in ["dashed", "dash", "dashes", "dashline"]:
                self._linestyle = Qt.DashLine
            else:
                self._linestyle = Qt.SolidLine  # default to solid
        return self._linestyle
    @property
    def textCharFormat(self):
        if self._textCharFormat is None:
            self._textCharFormat = QtGui.QTextCharFormat()
            self._textCharFormat.setForeground(self.fore)
            try:  # not all styles have a back property
                self._textCharFormat.setBackground(self.back)
            except Exception:
                pass
            self._textCharFormat.setUnderlineStyle(self.underline)
            if self.bold:
                self._textCharFormat.setFontWeight(QtGui.QFont.Bold)
            if self.italic:
                self._textCharFormat.setFontItalic(True)
        return self._textCharFormat
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
class TextReshaper:
    """Object to reshape a piece of text, taking indentation, paragraphs,
    comments and bulletpoints into account.
    """
    def __init__(self, lw, margin=3):
        self.lw = lw
        self.margin = margin
        self._lines1 = []
        self._lines2 = []
        self._wordBuffer = []
        self._charsInBuffer = -1  # First word ads one extra
        self._pendingPrefix = None  # A one-shot prefix
        self._currentPrefix = None  # The prefix used until a new prefix is set
    @classmethod
    def reshapeText(cls, text, lw):
        tr = cls(lw)
        tr.pushText(text)
        return tr.popText()
    def pushLine(self, line):
        """Push a single line to the input."""
        self._lines1.append(line.rstrip())
    def pushText(self, text):
        """Push a (multiline) text to the input."""
        for line in text.splitlines():
            self.pushLine(line)
    def popLines(self):
        """Get all available lines from the output."""
        try:
            while True:
                self._popLine()
        except StopIteration:
            self._flush()
        return [line for line in self._lines2]
    def popText(self):
        """Get all text from the output (i.e. lines joined with newline)."""
        return "\n".join(self.popLines())
    def _prefixString(self):
        if self._pendingPrefix is not None:
            prefix = self._pendingPrefix
            self._pendingPrefix = None
            return prefix
        else:
            return self._currentPrefix or ""
    def _addWordToBuffer(self, word):
        self._wordBuffer.append(word)
        self._charsInBuffer += len(word) + 1  # add one for space
    def _flush(self):
        if self._wordBuffer:
            self._lines2.append(self._prefixString() + "
".join(self._wordBuffer))
        self._wordBuffer, self._charsInBuffer = [], -1
    def _addNewParagraph(self):
        # Flush remaining words
        self._flush()
        # Create empty line
        prefix = self._currentPrefix or ""
        prefix = " " * len(prefix)
        self._lines2.append(prefix)
        # Allow new prefix
        self._currentPrefix = None
    def _popLine(self):
        """Pop a line from the input. Examine how it starts and convert it
        to words.
        """
        # Pop line
        try:
            line = self._lines1.pop(0)
        except IndexError:
            raise StopIteration()
        # Strip the line
        strippedline1 = line.lstrip()
        strippedline2 = line.lstrip(" \t#*")
        # Analyze this line (how does it start?)
        if not strippedline1:
            self._addNewParagraph()
            return
        elif strippedline1.startswith("* "):
            self._flush()
            indent = len(line) - len(strippedline1)
            linePrefix = line[:indent]
            self._pendingPrefix = linePrefix + "* "
            self._currentPrefix = linePrefix + "  "
        else:
            # Hey, an actual line! Determine prefix
            indent = len(line) - len(strippedline1)
            linePrefix = line[:indent]
            # Check comments
            if strippedline1.startswith("#"):
                linePrefix += "# "
            # What to do now?
            if linePrefix != self._currentPrefix:
                self._flush()
                self._currentPrefix = linePrefix
        # Process words one by one...
        for word in strippedline2.split(" "):
            self._addWordToBuffer(word)
            currentLineWidth = self._charsInBuffer + len(self._currentPrefix)
            if currentLineWidth < self.lw:
                # Not enough words in buffer yet
                pass
            elif len(self._wordBuffer) > 1:
                # Enough words to compose a line
                marginWith = currentLineWidth - self.lw
                marginWithout = self.lw - (currentLineWidth - len(word))
                if marginWith < marginWithout and marginWith < self.margin:
                    # add all buffered words
                    self._flush()
                else:
                    # add all buffered words (except last)
                    self._wordBuffer.pop(-1)
                    self._flush()
                    self._addWordToBuffer(word)
            else:
                # This single word covers more than one line
                self._flush()
testText = """
# This is a piece
# of comment
Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna
aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi
ut aliquip ex ea
commodo consequat. Duis aute irure dolor
in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
```

```
Excepteur sint occaecat cupidatat
non proident, sunt in culpa qui officia deserunt mollit anim
id est laborum.
        # Indented comments
        # should work
        # as well
skdb-a-very-long-word-
ksdbfksasdvbassdfhjsdfbjdfbvhjdbvhjbdfhjvbdfjbvjdfbvjdfbvjdbfvj
   A change in indentation makes it a separate line
sdckj bsdkjcb sdc
sdckj  foo bar
aap noot mies
  * Bullet points are preserved
  * Even if they are very long the should be preserved. I know that brevity is a
great virtue but you know,
    sometimes you just need those
    extra words to make a point.
"""
if __name__ == "__main__":
    print(TextReshaper.reshapeText(testText, 70))
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
""" This script runs a test for the code editor component.
"""
import os, sys
from qt import QtGui, QtCore, QtWidgets
Qt = QtCore.Qt
## Go up one directory and then import the codeeditor package
os.chdir("..")
sys.path.insert(0, ".")
from codeeditor import CodeEditor
if __name__ == "__main__":
    app = QtWidgets.QApplication([])
    # Create editor instance
    e = CodeEditor(
        highlightCurrentLine=True,
        longLineIndicatorPosition=20,
        showIndentationGuides=True,
        showWhitespace=True,
        showLineEndings=True,
        wrap=True,
        showLineNumbers=True,
    )
    QtWidgets.QShortcut(QtGui.QKeySequence("F1"), e).activated.connect(
        e.autocompleteShow
    )
    QtWidgets.QShortcut(QtGui.QKeySequence("F2"), e).activated.connect(
        e.autocompleteCancel
    )
    QtWidgets.QShortcut(QtGui.QKeySequence("F3"), e).activated.connect(
        lambda: e.calltipShow(0, "test(foo, bar)")
    )
    QtWidgets.QShortcut(QtGui.QKeySequence("Shift+Tab"), e).activated.connect(
        e.dedentSelection
    )  # Shift + Tab
    # TODO: somehow these shortcuts don't work in this test-app, but they do in
    # pyzo. May have something to do with overriding slots of Qt-native objects?
    QtWidgets.QShortcut(QtGui.QKeySequence("Ctrl+C"),
e).activated.connect(e.copy)
    QtWidgets.QShortcut(QtGui.QKeySequence("Ctrl+X"),
e).activated.connect(e.cut)
    QtWidgets.QShortcut(QtGui.QKeySequence("Ctrl+V"),
e).activated.connect(e.paste)
```

```
    QtWidgets.QShortcut(QtGui.QKeySequence("Ctrl+Shift+V"),
e).activated.connect(
        e.pasteAndSelect
    )
    QtWidgets.QShortcut(QtGui.QKeySequence("Ctrl+Z"),
e).activated.connect(e.undo)
    QtWidgets.QShortcut(QtGui.QKeySequence("Ctrl+Y"),
e).activated.connect(e.redo)
    e.setPlainText(
        "foo(bar)\nfor bar in range(5):\n  print bar\n"
        + "\nclass aap:\n  def monkey(self):\n    pass\n\n"
    )
    # Run application
    e.show()
    s = QtWidgets.QSplitter()
    s.addWidget(e)
    s.addWidget(QtWidgets.QLabel("test"))
    s.show()
    app.exec_()
```

```
# -*- coding: utf-8 -*-
# flake8: noqa
# Copyright (C) 2013, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" CodeEditor
A full featured code editor component based on QPlainTextEdit.
"""
from .manager import Manager
from .base import CodeEditorBase
from .extensions.appearance import (
    HighlightMatchingBracket,
    HighlightMatchingOccurrences,
    HighlightCurrentLine,
    FullUnderlines,
    IndentationGuides,
    CodeFolding,
    LongLineIndicator,
    ShowWhitespace,
    ShowLineEndings,
    Wrap,
    LineNumbers,
    SyntaxHighlighting,
    BreakPoints,
)
from .extensions.behaviour import (
    Indentation,
    HomeKey,
    EndKey,
    NumpadPeriodKey,
    AutoIndent,
    PythonAutoIndent,
    SmartCopyAndPaste,
    MoveLinesUpDown,
    ScrollWithUpDownKeys,
    AutoCloseQuotesAndBrackets,
)
from .extensions.autocompletion import AutoCompletion
from .extensions.calltip import Calltip
# Order of superclasses: first the extensions, then CodeEditorBase
# The first superclass is the first extension that gets to handle each key and
# the first to receive paint events.
```

```
class CodeEditor(
    HighlightCurrentLine,
    HighlightMatchingOccurrences,
    HighlightMatchingBracket,
    FullUnderlines,
    IndentationGuides,
    CodeFolding,
    LongLineIndicator,
    ShowWhitespace,
    ShowLineEndings,
    Wrap,
    BreakPoints,
    LineNumbers,
    AutoCompletion,  # Escape: first remove autocompletion,
    Calltip,  # then calltip
    Indentation,
    MoveLinesUpDown,
    ScrollWithUpDownKeys,
    HomeKey,
    EndKey,
    # NumpadPeriodKey,  -> disabled, see issue #720
    AutoIndent,
    PythonAutoIndent,
    AutoCloseQuotesAndBrackets,
    SyntaxHighlighting,
    SmartCopyAndPaste,  # overrides cut(), copy(), paste()
    CodeEditorBase,  # CodeEditorBase must be the last one in the list
):
    """
    CodeEditor with all the extensions
    """
    pass
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
"""
Code editor extensions that change its appearance
"""
from ..qt import QtGui, QtCore, QtWidgets
Qt = QtCore.Qt
from ..misc import ce_option
from ..manager import Manager
# todo: what about calling all extensions. CE_HighlightCurrentLine,
# or EXT_HighlightcurrentLine?
from ..parsers.tokens import ParenthesisToken
import enum
class HighlightMatchingOccurrences(object):
    # Register style element
    _styleElements = [
        (
            "Editor.Highlight matching occurrences",
            "The background color to highlight matching occurrences of the
currently selected word.",
            "back:#fdfda3",
        )
    ]
    def highlightMatchingOccurrences(self):
        """highlightMatchingOccurrences()
        Get whether to highlight matching occurrences.
        """
        return self.__highlightMatchingOccurrences
    @ce_option(True)
    def setHighlightMatchingOccurrences(self, value):
        """setHighlightMatchingOccurrences(value)
        Set whether to highlight matching occurrences.
        """
        self.__highlightMatchingOccurrences = bool(value)
        self.viewport().update()
    def _doHighlight(self, text):
        # make cursor at the beginning of the first visible block
        cursor = self.cursorForPosition(QtCore.QPoint(0, 0))
        doc = self.document()
        color =
self.getStyleElementFormat("editor.highlightMatchingOccurrences").back
        painter = QtGui.QPainter()
        painter.begin(self.viewport())
        painter.setBrush(color)
        painter.setPen(color.darker(110))
        # find occurrences
        for i in range(500):
            cursor = doc.find(
                text, cursor, doc.FindCaseSensitively | doc.FindWholeWords
            )
            if cursor is None or cursor.isNull():
                # no more matches
                break
            # don't highlight the actual selection
            if cursor == self.textCursor():
                continue
            endRect = self.cursorRect(cursor)
            if endRect.bottom() > self.height():
                # rest of document is not visible, don't bother highlighting
                break
            cursor.setPosition(min(cursor.position(), cursor.anchor()))
            startRect = self.cursorRect(cursor)
            width = endRect.left() - startRect.left()
            painter.drawRect(
                startRect.left(), startRect.top(), width, startRect.height()
            )
            # move to end of word again, otherwise we never advance in the doc
            cursor.movePosition(cursor.EndOfWord)
        else:
            print("Matching selection highlighting did not break")
        painter.end()
    def paintEvent(self, event):
        """paintEvent(event)
        If there is a current selection, and the selected text is a valid Python
        identifier (no whitespace, starts with a letter), then highlight all the
        matching occurrences of the selected text in the current view.
        Paints behinds its super().
        """
        cursor = self.textCursor()
        if self.__highlightMatchingOccurrences and cursor.hasSelection():
            text = cursor.selectedText()
            if text.isidentifier():
                self._doHighlight(text)
        super(HighlightMatchingOccurrences, self).paintEvent(event)
class _ParenNotFound(Exception):
    pass
class _ParenIterator:
    """Iterates in given direction over parentheses in the document.
    Uses the stored token-list of the blocks.
    Iteration gives both a parenthesis and its global position."""
    def __init__(self, cursor, direction):
        self.cur_block = cursor.block()
        self.cur_tokens = self._getParenTokens()
        self.direction = direction
        # We need to know where we start in the current token list
        k = 0
        try:
            while self.cur_tokens[k].end != cursor.positionInBlock():
                k += 1
            self.cur_pos = k
        except IndexError:
            # If the parenthesis cannot be found, it means that it is not
inluded
            # in any token, ie. it is part of a string or comment
            raise _ParenNotFound
    def _getParenTokens(self):
        try:
            return list(
                filter(
                    lambda x: isinstance(x, ParenthesisToken),
                    self.cur_block.userData().tokens,
                )
            )
        except AttributeError:
            return (
                []
            )  # can be a piece of text that we do not tokenize or have not
stored tokens
    def __iter__(self):
        return self
    def __next__(self):
        self.cur_pos += self.direction
        while self.cur_pos >= len(self.cur_tokens) or self.cur_pos < 0:
            if self.direction == 1:
                self.cur_block = self.cur_block.next()
            else:
                self.cur_block = self.cur_block.previous()
            if not self.cur_block.isValid():
                raise StopIteration
            self.cur_tokens = self._getParenTokens()
            if self.direction == 1:
                self.cur_pos = 0
            else:
                self.cur_pos = len(self.cur_tokens) - 1
        return (
            self.cur_tokens[self.cur_pos]._style,
            self.cur_block.position() + self.cur_tokens[self.cur_pos].end,
        )
class _PlainTextParenIterator:
    """Iterates in given direction over parentheses in the document.
    To be used when there is no parser.
    Iteration gives both a parenthesis and its global position."""
    def __init__(self, cursor, direction):
        self.fulltext = cursor.document().toPlainText()
        self.position = cursor.position() - 1
        self.direction = direction
    def __iter__(self):
        return self
    def __next__(self):
        self.position += self.direction
        try:
            while self.fulltext[self.position] not in "([{)]}":
                self.position += self.direction
                if self.position < 0:
                    raise StopIteration
        except IndexError:
            raise StopIteration
        return self.fulltext[self.position], self.position + 1
class _MatchStatus(enum.Enum):
    NoMatch = 0
    Match = 1
    MisMatch = 2
class _MatchResult:
    def __init__(self, status, corresponding=None, offending=None):
        self.status = status
        self.corresponding = corresponding
        self.offending = offending
class HighlightMatchingBracket(object):
    # Register style element
    _styleElements = [
        (
            "Editor.Highlight matching bracket",
            "The background color to highlight matching brackets.",
            "back:#ccc",
        ),
        (
            "Editor.Highlight unmatched bracket",
            "The background color to highlight unmatched brackets.",
            "back:#F7BE81",
        ),
        (
            "Editor.Highlight mismatching bracket",
            "The background color to highlight mismatching brackets.",
            "back:#F7819F",
        ),
    ]
    def highlightMatchingBracket(self):
        """highlightMatchingBracket()
        Get whether to highlight matching brackets.
        """
        return self.__highlightMatchingBracket
    @ce_option(True)
    def setHighlightMatchingBracket(self, value):
        """setHighlightMatchingBracket(value)
        Set whether to highlight matching brackets.
        """
        self.__highlightMatchingBracket = bool(value)
        self.viewport().update()
    def highlightMisMatchingBracket(self):
        """highlightMisMatchingBracket()
        Get whether to highlight mismatching brackets.
        """
        return self.__highlightMisMatchingBracket
    @ce_option(True)
    def setHighlightMisMatchingBracket(self, value):
        """setHighlightMisMatchingBracket(value)
        Set whether to highlight mismatching brackets.
        """
        self.__highlightMisMatchingBracket = bool(value)
        self.viewport().update()
    def _highlightSingleChar(self, painter, cursor, width, colorname):
        """_highlightSingleChar(painter, cursor, width, colorname)
        Draws a highlighting rectangle around the single character to the
        left of the specified cursor.
        """
        cursor_rect = self.cursorRect(cursor)
        top = cursor_rect.top()
        left = cursor_rect.left() - width
        height = cursor_rect.bottom() - top + 1
        color = self.getStyleElementFormat(colorname).back
        painter.setBrush(color)
        painter.setPen(color.darker(110))
        painter.drawRect(QtCore.QRect(int(left), int(top), int(width),
int(height)))
    _matchingBrackets = {"(": ")", "[": "]", "{": "}", ")": "(", "]": "[", "}":
"{"}
    def _findMatchingBracket(self, char, cursor):
        """_findMatchingBracket(char, cursor)
        Find a bracket that matches the specified char in the specified
document.
        Return a _MatchResult object indicating whether this succeded and the
        positions of the parentheses causing this result.
        """
        if char in ")]}":
            direction = -1
            stacking = ")]}"
            unstacking = "([{"
        elif char in "([{":
            direction = 1
            stacking = "([{"
            unstacking = ")]}"
        else:
            raise ValueError("invalid bracket character: " + char)
        stacked_paren = [(char, cursor.position())]  # using a Python list as a
stack
        # stack not empty because the _ParenIterator will not give back
        # the parenthesis we're matching
        our_iterator = (
            _ParenIterator
            if self.parser() is not None and self.parser().name() != ""
            else _PlainTextParenIterator
        )
        for paren, pos in our_iterator(cursor, direction):
            if paren in stacking:
                stacked_paren.append((paren, pos))
            elif paren in unstacking:
                if self._matchingBrackets[stacked_paren[-1][0]] != paren:
                    return _MatchResult(
                        _MatchStatus.MisMatch, pos, stacked_paren[-1][1]
                    )
                else:
                    stacked_paren.pop()
            if len(stacked_paren) == 0:
                # we've found our match
                return _MatchResult(_MatchStatus.Match, pos)
        return _MatchResult(_MatchStatus.NoMatch)
    def _cursorAt(self, doc, pos):
        new_cursor = QtGui.QTextCursor(doc)
        new_cursor.setPosition(pos)
        return new_cursor
    def paintEvent(self, event):
        """paintEvent(event)
        If the current cursor is positioned to the right of a bracket ()[]{},
        look for a matching one, and, if found, draw a highlighting rectangle
        around both brackets of the pair.
        Paints behinds its super().
        """
        if not self.__highlightMatchingBracket:
            super(HighlightMatchingBracket, self).paintEvent(event)
            return
        cursor = QtGui.QTextCursor(self.textCursor())
        if cursor.atBlockStart():
            cursor.movePosition(cursor.Right)
            movedRight = True
        else:
            movedRight = False
        text = cursor.block().text()
        pos = cursor.positionInBlock() - 1
        if len(text) > pos and len(text) > 0:
            # get the character to the left of the cursor
            char = text[pos]
            if not movedRight and char not in "()[]{}" and len(text) > pos + 1:
                # no brace to the left of cursor; try to the right
                cursor.movePosition(cursor.Right)
                char = text[pos + 1]
            if char in "()[]{}":
                doc = cursor.document()
                try:
                    match_res = self._findMatchingBracket(char, cursor)
                    fm = QtGui.QFontMetrics(doc.defaultFont())
                    width = fm.width(
                        char
                    )  # assumes that both paren have the same width
                    painter = QtGui.QPainter()
                    painter.begin(self.viewport())
                    if match_res.status == _MatchStatus.NoMatch:
                        self._highlightSingleChar(
                            painter, cursor, width,
"editor.highlightUnmatchedBracket"
                        )
                    elif match_res.status == _MatchStatus.Match:
                        self._highlightSingleChar(
                            painter, cursor, width,
"editor.highlightMatchingBracket"
                        )
                        self._highlightSingleChar(
                            painter,
                            self._cursorAt(doc, match_res.corresponding),
                            width,
                            "editor.highlightMatchingBracket",
                        )
                    else:  # this is a mismatch
                        if (
                            cursor.position() != match_res.offending
                            or not self.highlightMisMatchingBracket()
                        ):
                            self._highlightSingleChar(
                                painter,
                                cursor,
                                width,
                                "editor.highlightUnmatchedBracket",
                            )
                        if self.highlightMisMatchingBracket():
                            self._highlightSingleChar(
                                painter,
                                self._cursorAt(doc, match_res.corresponding),
                                width,
                                "editor.highlightMisMatchingBracket",
                            )
                            self._highlightSingleChar(
                                painter,
                                self._cursorAt(doc, match_res.offending),
                                width,
                                "editor.highlightMisMatchingBracket",
                            )
                    painter.end()
                except _ParenNotFound:
                    # is raised when current parenthesis is not
                    # found in its line token list, meaning it is in a string
literal
                    pass
        super(HighlightMatchingBracket, self).paintEvent(event)
class HighlightCurrentLine(object):
    """
    Highlight the current line
    """
    # Register style element
    _styleElements = [
        (
            "Editor.Highlight current line",
            "The background color of the current line highlight.",
            "back:#ffff99",
        )
    ]
    def highlightCurrentLine(self):
        """highlightCurrentLine()
        Get whether to highlight the current line.
        """
        return self.__highlightCurrentLine
    @ce_option(True)
    def setHighlightCurrentLine(self, value):
        """setHighlightCurrentLine(value)
        Set whether to highlight the current line.
        """
        self.__highlightCurrentLine = bool(value)
        self.viewport().update()
    def paintEvent(self, event):
        """paintEvent(event)
        Paints a rectangle spanning the current block (in case of line wrapping,
this
        means multiple lines)
        Paints behind its super()
        """
        if not self.highlightCurrentLine():
            super(HighlightCurrentLine, self).paintEvent(event)
            return
        # Get color
        color = self.getStyleElementFormat("editor.highlightCurrentLine").back
        # Find the top of the current block, and the height
        cursor = self.textCursor()
        cursor.movePosition(cursor.StartOfBlock)
        top = self.cursorRect(cursor).top()
        cursor.movePosition(cursor.EndOfBlock)
        height = self.cursorRect(cursor).bottom() - top + 1
        margin = self.document().documentMargin()
        painter = QtGui.QPainter()
        painter.begin(self.viewport())
        painter.fillRect(
            QtCore.QRect(
                int(margin),
                int(top),
                int(self.viewport().width() - 2 * margin),
                int(height),
            ),
            color,
        )
        painter.end()
        super(HighlightCurrentLine, self).paintEvent(event)
        # for debugging paint events
        # if 'log' not in self.__class__.__name__.lower():
        #    print(height, event.rect().width())
class IndentationGuides(object):
    # Register style element
    _styleElements = [
        (
            "Editor.Indentation guides",
            "The color and style of the indentation guides.",
            "fore:#DDF,linestyle:solid",
        )
    ]
    def showIndentationGuides(self):
        """showIndentationGuides()
        Get whether to show indentation guides.
        """
        return self.__showIndentationGuides
    @ce_option(True)
    def setShowIndentationGuides(self, value):
        """setShowIndentationGuides(value)
        Set whether to show indentation guides.
        """
        self.__showIndentationGuides = bool(value)
        self.viewport().update()
    def paintEvent(self, event):
        """paintEvent(event)
        Paint the indentation guides, using the indentation info calculated
        by the highlighter.
        """
        super(IndentationGuides, self).paintEvent(event)
        if not self.showIndentationGuides():
            return
        # Get doc and viewport
        doc = self.document()
        viewport = self.viewport()
        # Get multiplication factor and indent width
        indentWidth = self.indentWidth()
        if self.indentUsingSpaces():
            factor = 1
        else:
            factor = indentWidth
        # Init painter
        painter = QtGui.QPainter()
        painter.begin(viewport)
        # Prepare pen
        format = self.getStyleElementFormat("editor.IndentationGuides")
        pen = QtGui.QPen(format.fore)
        pen.setStyle(format.linestyle)
        painter.setPen(pen)
        offset = doc.documentMargin() + self.contentOffset().x()
        def paintIndentationGuides(cursor):
            y3 = self.cursorRect(cursor).top()
            y4 = self.cursorRect(cursor).bottom()
            bd = cursor.block().userData()
            if bd and hasattr(bd, "indentation") and bd.indentation:
                for x in range(indentWidth, bd.indentation * factor,
indentWidth):
                    w = self.fontMetrics().width("i" * x) + offset
                    w += 1  # Put it more under the block
                    if w > 0:  # if scrolled horizontally it can become < 0
                        painter.drawLine(QtCore.QLine(int(w), int(y3), int(w),
int(y4)))
```

```
        self.doForVisibleBlocks(paintIndentationGuides)
        # Done
        painter.end()
class FullUnderlines(object):
    def paintEvent(self, event):
        """paintEvent(event)
        Paint a horizontal line for the blocks for which there is a
        syntax format that has underline:full. Whether this is the case
        is stored at the blocks user data.
        """
        super(FullUnderlines, self).paintEvent(event)
        painter = QtGui.QPainter()
        painter.begin(self.viewport())
        margin = self.document().documentMargin()
        w = self.viewport().width()
        def paintUnderline(cursor):
            y = self.cursorRect(cursor).bottom()
            bd = cursor.block().userData()
            try:
                fullUnderlineFormat = bd.fullUnderlineFormat
            except AttributeError:
                pass  # fullUnderlineFormat may not be an attribute
            else:
                if fullUnderlineFormat is not None:
                    # Apply pen
                    pen = QtGui.QPen(fullUnderlineFormat.fore)
                    pen.setStyle(fullUnderlineFormat.linestyle)
                    painter.setPen(pen)
                    # Paint
                    painter.drawLine(
                        QtCore.QLine(int(margin), int(y), int(w - 2 * margin),
int(y))
                    )
        self.doForVisibleBlocks(paintUnderline)
        painter.end()
class CodeFolding(object):
    def paintEvent(self, event):
        """paintEvent(event)"""
        super(CodeFolding, self).paintEvent(event)
        return  # Code folding code is not yet complete
        painter = QtGui.QPainter()
        painter.begin(self.viewport())
        margin = int(self.document().documentMargin())
        def paintCodeFolders(cursor):
            y = int(self.cursorRect(cursor).top())
            h = int(self.cursorRect(cursor).height())
            rect = QtCore.QRect(margin, y, h, h)
            text = cursor.block().text()
            if text.rstrip().endswith(":"):
                painter.drawRect(rect)
                painter.drawText(
                    rect, QtCore.Qt.AlignVCenter | QtCore.Qt.AlignHCenter, "-"
                )
                # Apply pen
                # Paint
                # painter.drawLine(QtCore.QLine(int(margin), int(y), int(w -
2*margin), int(y)))
        self.doForVisibleBlocks(paintCodeFolders)
        painter.end()
class LongLineIndicator(object):
    # Register style element
    _styleElements = [
        (
            "Editor.Long line indicator",
            "The color and style of the long line indicator.",
            "fore:#BBB,linestyle:solid",
        )
    ]
    def longLineIndicatorPosition(self):
        """longLineIndicatorPosition()
        Get the position of the long line indicator (aka edge column).
        A value of 0 or smaller means that no indicator is shown.
        """
        return self.__longLineIndicatorPosition
    @ce_option(80)
    def setLongLineIndicatorPosition(self, value):
        """setLongLineIndicatorPosition(value)
        Set the position of the long line indicator (aka edge column).
        A value of 0 or smaller means that no indicator is shown.
        """
        self.__longLineIndicatorPosition = int(value)
        self.viewport().update()
    def paintEvent(self, event):
        """paintEvent(event)
        Paint the long line indicator. Paints behind its super()
        """
        if self.longLineIndicatorPosition() <= 0:
            super(LongLineIndicator, self).paintEvent(event)
            return
        # Get doc and viewport
        doc = self.document()
        viewport = self.viewport()
        # Get position of long line
        fm = self.fontMetrics()
        # width of ('i'*length) not length * (width of 'i') b/c of
        # font kerning and rounding
        x = fm.width("i" * self.longLineIndicatorPosition())
        x += doc.documentMargin() + self.contentOffset().x()
        x += 1  # Move it a little next to the cursor
        # Prepate painter
        painter = QtGui.QPainter()
        painter.begin(viewport)
        # Prepare pen
        format = self.getStyleElementFormat("editor.LongLineIndicator")
        pen = QtGui.QPen(format.fore)
        pen.setStyle(format.linestyle)
        painter.setPen(pen)
        # Draw line and end painter
        painter.drawLine(QtCore.QLine(int(x), 0, int(x),
int(viewport.height())))
        painter.end()
        # Propagate event
        super(LongLineIndicator, self).paintEvent(event)
class ShowWhitespace(object):
    def showWhitespace(self):
        """Show or hide whitespace markers"""
        option = self.document().defaultTextOption()
        return bool(option.flags() & option.ShowTabsAndSpaces)
    @ce_option(False)
    def setShowWhitespace(self, value):
        try:
            option = self.document().defaultTextOption()
            if value:
                option.setFlags(option.flags() | option.ShowTabsAndSpaces)
            else:
                option.setFlags(option.flags() & ~option.ShowTabsAndSpaces)
            self.document().setDefaultTextOption(option)
        except Exception:
            # This can produce: 2147483617 is not a valid QTextOption.Flag
            # and I do not know how to avoid it :/
            pass
class ShowLineEndings(object):
    @ce_option(False)
    def showLineEndings(self):
        """Get whether line ending markers are shown."""
        option = self.document().defaultTextOption()
        return bool(option.flags() & option.ShowLineAndParagraphSeparators)
    def setShowLineEndings(self, value):
        try:
            option = self.document().defaultTextOption()
            if value:
                option.setFlags(option.flags() |
option.ShowLineAndParagraphSeparators)
            else:
                option.setFlags(option.flags() &
~option.ShowLineAndParagraphSeparators)
            self.document().setDefaultTextOption(option)
        except Exception:
            # This can produce: 2147483617 is not a valid QTextOption.Flag
            # and I do not know how to avoid it :/
            pass
class LineNumbers(object):
    # Margin on both side of the line numbers
    _LineNumberAreaMargin = 3
    # Register style element
    _styleElements = [
        (
            "Editor.Line numbers",
            "The text- and background-color of the line numbers.",
            "fore:#222,back:#DDD",
        )
    ]
    class __LineNumberArea(QtWidgets.QWidget):
        """This is the widget reponsible for drawing the line numbers."""
        def __init__(self, codeEditor):
            QtWidgets.QWidget.__init__(self, codeEditor)
            self.setCursor(QtCore.Qt.PointingHandCursor)
            self._pressedY = None
            self._lineNrChoser = None
        def _getY(self, pos):
            tmp = self.mapToGlobal(pos)
            return self.parent().viewport().mapFromGlobal(tmp).y()
        def mousePressEvent(self, event):
            self._pressedY = self._getY(event.pos())
        def mouseReleaseEvent(self, event):
            self._handleWholeBlockSelection(self._getY(event.pos()))
        def mouseMoveEvent(self, event):
            self._handleWholeBlockSelection(self._getY(event.pos()))
        def _handleWholeBlockSelection(self, y2):
            # Get y1 and sort (y1, y2)
            y1 = self._pressedY
            if y1 is None:
                y1 = y2
            y1, y2 = min(y1, y2), max(y1, y2)
            # Get cursor and two cursors corresponding to selected blocks
            editor = self.parent()
            cursor = editor.textCursor()
            c1 = editor.cursorForPosition(QtCore.QPoint(0, int(y1)))
            c2 = editor.cursorForPosition(QtCore.QPoint(0, int(y2)))
            # Make these two cursors select the whole block
            c1.movePosition(c1.StartOfBlock, c1.MoveAnchor)
            c2.movePosition(c2.EndOfBlock, c2.MoveAnchor)
            # Apply selection
            cursor.setPosition(c1.position(), cursor.MoveAnchor)
            cursor.setPosition(c2.position(), cursor.KeepAnchor)
            editor.setTextCursor(cursor)
        def mouseDoubleClickEvent(self, event):
            self.showLineNumberChoser()
        def showLineNumberChoser(self):
            # Create line number choser if needed
            if self._lineNrChoser is None:
                self._lineNrChoser = LineNumbers.LineNumberChoser(self.parent())
            # Get editor and cursor
            editor = self.parent()
            cursor = editor.textCursor()
            # Get (x,y) pos and apply
            x, y = self.width() + 4, editor.cursorRect(cursor).y()
            self._lineNrChoser.move(int(x), int(y))
            # Show/reset line number choser
            self._lineNrChoser.reset(cursor.blockNumber() + 1)
        def paintEvent(self, event):
            editor = self.parent()
            if not editor.showLineNumbers():
                return
            # Get doc and viewport
            viewport = editor.viewport()
            # Get format and margin
            format = editor.getStyleElementFormat("editor.LineNumbers")
            margin = editor._LineNumberAreaMargin
            # Init painter
            painter = QtGui.QPainter()
            painter.begin(self)
            # Get which part to paint. Just do all to avoid glitches
            w = editor.getLineNumberAreaWidth()
            y1, y2 = 0, editor.height()
            # y1, y2 = event.rect().top()-10, event.rect().bottom()+10
            # Get offset
            tmp = self.mapToGlobal(QtCore.QPoint(0, 0))
            offset = viewport.mapFromGlobal(tmp).y()
            # Draw the background
            painter.fillRect(QtCore.QRect(0, int(y1), int(w), int(y2)),
format.back)
            # Get cursor
            cursor = editor.cursorForPosition(QtCore.QPoint(0, int(y1)))
            # Prepare fonts
            font1 = editor.font()
            font2 = editor.font()
            font2.setBold(True)
            currentBlockNumber = editor.textCursor().block().blockNumber()
            # Init painter with font and color
            painter.setFont(font1)
            painter.setPen(format.fore)
            # Repainting always starts at the first block in the viewport,
            # regardless of the event.rect().y(). Just to keep it simple
            while True:
                blockNumber = cursor.block().blockNumber()
                y = editor.cursorRect(cursor).y()
                # Set font to bold if line number is the current
                if blockNumber == currentBlockNumber:
                    painter.setFont(font2)
                painter.drawText(
                    0, y - offset, w - margin, 50, Qt.AlignRight,
str(blockNumber + 1)
                )
                # Set font back
                if blockNumber == currentBlockNumber:
                    painter.setFont(font1)
                if y > y2:
                    break  # Reached end of the repaint area
                if not cursor.block().next().isValid():
                    break  # Reached end of the text
                cursor.movePosition(cursor.NextBlock)
            # Done
            painter.end()
    class LineNumberChoser(QtWidgets.QSpinBox):
        def __init__(self, parent):
            QtWidgets.QSpinBox.__init__(self, parent)
            self._editor = parent
            ss = (
                "QSpinBox { border: 2px solid #789; border-radius: 3px; padding: 
4px; }"
            )
            self.setStyleSheet(ss)
            self.setPrefix("Go to line: ")
            self.setAccelerated(True)
            self.setButtonSymbols(self.NoButtons)
            self.setCorrectionMode(self.CorrectToNearestValue)
            # Signal for when value changes, and flag to disbale it once
            self._ignoreSignalOnceFlag = False
            self.valueChanged.connect(self.onValueChanged)
        def reset(self, currentLineNumber):
            # Set value to (given) current line number
            self._ignoreSignalOnceFlag = True
            self.setRange(1, self._editor.blockCount())
            self.setValue(currentLineNumber)
            # Select text and focus so that the user can simply start typing
            self.selectAll()
            self.setFocus()
            # Make visible
            self.show()
            self.raise_()
        def focusOutEvent(self, event):
            self.hide()
        def keyPressEvent(self, event):
            if event.key() in [
                QtCore.Qt.Key_Escape,
                QtCore.Qt.Key_Enter,
                QtCore.Qt.Key_Return,
            ]:
                self._editor.setFocus()  # Moves focus away, thus hiding self
            else:
                QtWidgets.QSpinBox.keyPressEvent(self, event)
        def onValueChanged(self, nr):
            if self._ignoreSignalOnceFlag:
                self._ignoreSignalOnceFlag = False
            else:
                self._editor.gotoLine(nr)
    def __init__(self, *args, **kwds):
        self.__lineNumberArea = None
        super(LineNumbers, self).__init__(*args, **kwds)
        # Create widget that draws the line numbers
        self.__lineNumberArea = self.__LineNumberArea(self)
        # Issue an update when the font or amount of line numbers changes
        self.blockCountChanged.connect(self.__onBlockCountChanged)
        self.fontChanged.connect(self.__onBlockCountChanged)
        self.__onBlockCountChanged()
        self.addLeftMargin(LineNumbers, self.getLineNumberAreaWidth)
    def gotoLinePopup(self):
        """Popup the little widget to quickly goto a certain line.
        Can also be achieved by double-clicking the line number area.
        """
        self.__lineNumberArea.showLineNumberChoser()
    def showLineNumbers(self):
        return self.__showLineNumbers
    @ce_option(True)
    def setShowLineNumbers(self, value):
        self.__showLineNumbers = bool(value)
        # Note that this method is called before the __init__ is finished,
        # so that the __lineNumberArea is not yet created.
        if self.__lineNumberArea:
            if self.__showLineNumbers:
                self.__onBlockCountChanged()
                self.__lineNumberArea.show()
            else:
                self.__lineNumberArea.hide()
            self.updateMargins()
    def getLineNumberAreaWidth(self):
        """
        Count the number of lines, compute the length of the longest line number
        (in pixels)
        """
        if not self.__showLineNumbers:
            return 0
        lastLineNumber = self.blockCount()
        margin = self._LineNumberAreaMargin
        return self.fontMetrics().width(str(lastLineNumber)) + 2 * margin
    def __onBlockCountChanged(self, count=None):
        """
        Update the line number area width. This requires to set the
        viewport margins, so there is space to draw the linenumber area
        """
        if self.__showLineNumbers:
            self.updateMargins()
    def resizeEvent(self, event):
        super(LineNumbers, self).resizeEvent(event)
        # On resize, resize the lineNumberArea, too
        rect = self.contentsRect()
        m = self.getLeftMargin(LineNumbers)
        w = self.getLineNumberAreaWidth()
        self.__lineNumberArea.setGeometry(rect.x() + m, rect.y(), w,
rect.height())
    def paintEvent(self, event):
        super(LineNumbers, self).paintEvent(event)
        # On repaint, update the complete line number area
        w = self.getLineNumberAreaWidth()
        self.__lineNumberArea.update(0, 0, w, self.height())
class BreakPoints(object):
    _breakPointWidth = 11  # With of total bar, actual points are smaller
    # Register style element
    _styleElements = [
        (
            "Editor.BreakPoints",
            "The fore- and background-color of the breakpoints.",
            "fore:#F66,back:#dfdfe1",
        )
    ]
    class __BreakPointArea(QtWidgets.QWidget):
        """This is the widget reponsible for drawing the break points."""
        def __init__(self, codeEditor):
            QtWidgets.QWidget.__init__(self, codeEditor)
            self.setCursor(QtCore.Qt.PointingHandCursor)
            self.setMouseTracking(True)
            self._virtualBreakpoint = 0
        def _getY(self, pos):
            tmp = self.mapToGlobal(pos)
            return self.parent().viewport().mapFromGlobal(tmp).y()
        def mousePressEvent(self, event):
            self._toggleBreakPoint(self._getY(event.pos()))
        def mouseMoveEvent(self, event):
            y = self._getY(event.pos())
            editor = self.parent()
            c1 = editor.cursorForPosition(QtCore.QPoint(0, int(y)))
            self._virtualBreakpoint = c1.blockNumber() + 1
            self.update()
        def leaveEvent(self, event):
            self._virtualBreakpoint = 0
            self.update()
        def _toggleBreakPoint(self, y):
            # Get breakpoint corresponding to pressed pos
            editor = self.parent()
            c1 = editor.cursorForPosition(QtCore.QPoint(0, int(y)))
            linenr = c1.blockNumber() + 1
            # Toggle
            self.parent().toggleBreakpoint(linenr)
        def paintEvent(self, event):
            editor = self.parent()
            if not editor.showBreakPoints():
                return
            # Get format and margin
            format = editor.getStyleElementFormat("editor.breakpoints")
            margin = 1
            w = editor._breakPointWidth
            bulletWidth = w - 2 * margin
            # Init painter
            painter = QtGui.QPainter()
            painter.begin(self)
            # Get which part to paint. Just do all to avoid glitches
            y1, y2 = 0, editor.height()
            # Draw the background
            painter.fillRect(QtCore.QRect(0, int(y1), int(w), int(y2)),
format.back)
            # Get debug indicator and list of sorted breakpoints
            debugBlockIndicator = editor._debugLineIndicator - 1
            virtualBreakpoint = self._virtualBreakpoint - 1
            blocknumbers = [i - 1 for i in sorted(self.parent()._breakPoints)]
            if not (
                blocknumbers
                or editor._debugLineIndicator
                or editor._debugLineIndicators
                or virtualBreakpoint > 0
            ):
                return
            # Get cursor
            cursor = editor.cursorForPosition(QtCore.QPoint(0, int(y1)))
            # Get start block number and bullet offset in pixels
            startBlockNumber = cursor.block().blockNumber()
            bulletOffset = editor.contentOffset().y() + bulletWidth * 0.25
            # Prepare painter
            painter.setPen(QtGui.QColor("#777"))
            painter.setBrush(format.fore)
            painter.setRenderHint(painter.Antialiasing)
            # Draw breakpoints
            for blockNumber in blocknumbers:
                if blockNumber < startBlockNumber:
                    continue
                # Get block
                block = editor.document().findBlockByNumber(blockNumber)
                if block.isValid():
                    y = editor.blockBoundingGeometry(block).y() + bulletOffset
                    painter.drawEllipse(
                        int(margin), int(y), int(bulletWidth), int(bulletWidth)
                    )
            # Draw *the* debug marker
            if debugBlockIndicator > 0:
                painter.setBrush(QtGui.QColor("#6F6"))
                # Get block
                block = editor.document().findBlockByNumber(debugBlockIndicator)
                if block.isValid():
                    y = editor.blockBoundingGeometry(block).y() + bulletOffset
                    y += 0.25 * bulletWidth
                    painter.drawEllipse(
                        int(margin), int(y), int(bulletWidth), int(0.5 *
bulletWidth)
                    )
            # Draw other debug markers
            for debugLineIndicator in editor._debugLineIndicators:
                debugBlockIndicator = debugLineIndicator - 1
                painter.setBrush(QtGui.QColor("#DDD"))
                # Get block
                block = editor.document().findBlockByNumber(debugBlockIndicator)
                if block.isValid():
                    y = editor.blockBoundingGeometry(block).y() + bulletOffset
                    y += 0.25 * bulletWidth
                    painter.drawEllipse(
                        int(margin), int(y), int(bulletWidth), int(0.5 *
bulletWidth)
                    )
            # Draw virtual break point
            if virtualBreakpoint > 0:
                painter.setBrush(QtGui.QColor(0, 0, 0, 0))
                # Get block
                block = editor.document().findBlockByNumber(virtualBreakpoint)
                if block.isValid():
                    y = editor.blockBoundingGeometry(block).y() + bulletOffset
                    painter.drawEllipse(
                        int(margin), int(y), int(bulletWidth), int(bulletWidth)
                    )
            # Done
            painter.end()
    def __init__(self, *args, **kwds):
        self.__breakPointArea = None
        super(BreakPoints, self).__init__(*args, **kwds)
        # Create widget that draws the breakpoints
        self.__breakPointArea = self.__BreakPointArea(self)
        self.addLeftMargin(BreakPoints, self.getBreakPointAreaWidth)
        self._breakPoints = {}  # int -> block
        self._debugLineIndicator = 0
        self._debugLineIndicators = set()
        self.blockCountChanged.connect(self.__onBlockCountChanged)
    def __onBlockCountChanged(self):
        """Track breakpoints so we can update the number when text is inserted
        above.
        """
        newBreakPoints = {}
        for linenr in list(self._breakPoints):
            block, block_previous, block_next = self._breakPoints[linenr]
            block_linenr = block.blockNumber() + 1
            prev_ok = block.previous().blockNumber() ==
block_previous.blockNumber()
            next_ok = block.next().blockNumber() == block_next.blockNumber()
            if prev_ok or next_ok:
                if block_linenr == linenr:
                    if prev_ok and next_ok:
                        pass  # All is well
                    else:
                        # Update refs
                        self._breakPoints[linenr] = (
                            block,
                            block.previous(),
                            block.next(),
                        )
                else:
                    # Update linenr - this is the only case where "move" th bp
                    newBreakPoints[block_linenr] = self._breakPoints.pop(linenr)
            else:
                if block_linenr == linenr:
                    # Just update refs
                    self._breakPoints[linenr] = block, block.previous(),
block.next()
                else:
                    # Delete breakpoint? Meh, just update refs
                    self._breakPoints[linenr] = block, block.previous(),
block.next()
        if newBreakPoints:
            self._breakPoints.update(newBreakPoints)
            self.breakPointsChanged.emit(self)
            self.__breakPointArea.update()
    def breakPoints(self):
        """A list of breakpoints for this editor."""
        return list(sorted(self._breakPoints))
    def clearBreakPoints(self):
        """Remove all breakpoints for this editor."""
        for linenr in self.breakPoints():
            self.toggleBreakpoint(linenr)
    def toggleBreakpoint(self, linenr=None):
        """Turn breakpoint on/off for given linenr of current line."""
        if linenr is None:
            linenr = self.textCursor().blockNumber() + 1
        if linenr in self._breakPoints:
            self._breakPoints.pop(linenr)
        else:
            c = self.textCursor()
            c.movePosition(c.Start)
            c.movePosition(c.NextBlock, c.MoveAnchor, linenr - 1)
            b = c.block()
            self._breakPoints[linenr] = b, b.previous(), b.next()
        self.breakPointsChanged.emit(self)
        self.__breakPointArea.update()
    def setDebugLineIndicator(self, linenr, active=True):
        """Set the debug line indicator to the given line number.
        If None or 0, the indicator is hidden.
        """
        linenr = int(linenr or 0)
        if not linenr:
            # Remove all indicators
            if self._debugLineIndicator or self._debugLineIndicators:
                self._debugLineIndicator = 0
                self._debugLineIndicators = set()
                self.__breakPointArea.update()
        elif active:
            # Set *the* indicator
            if linenr != self._debugLineIndicator:
                self._debugLineIndicators.discard(linenr)
                self._debugLineIndicator = linenr
                self.__breakPointArea.update()
        else:
            # Add to set of indicators
            if linenr not in self._debugLineIndicators:
                self._debugLineIndicators.add(linenr)
                self.__breakPointArea.update()
    def getBreakPointAreaWidth(self):
        if not self.__showBreakPoints:
            return 0
        else:
            return self._breakPointWidth
    def showBreakPoints(self):
        return self.__showBreakPoints
    @ce_option(True)
    def setShowBreakPoints(self, value):
        self.__showBreakPoints = bool(value)
        # Note that this method is called before the __init__ is finished,
        # so that the area is not yet created.
        if self.__breakPointArea:
            if self.__showBreakPoints:
                self.__breakPointArea.show()
            else:
                self.__breakPointArea.hide()
                self.clearBreakPoints()
            self.updateMargins()
    def resizeEvent(self, event):
        super(BreakPoints, self).resizeEvent(event)
        # On resize, resize the breakpointArea, too
        rect = self.contentsRect()
        m = self.getLeftMargin(BreakPoints)
        w = self.getBreakPointAreaWidth()
        self.__breakPointArea.setGeometry(rect.x() + m, rect.y(), w,
rect.height())
    def paintEvent(self, event):
        super(BreakPoints, self).paintEvent(event)
        # On repaint, update the complete breakPointArea
        w = self.getBreakPointAreaWidth()
        self.__breakPointArea.update(0, 0, w, self.height())
class Wrap(object):
    def wrap(self):
        """Enable or disable wrapping"""
        option = self.document().defaultTextOption()
        return not bool(option.wrapMode() == option.NoWrap)
    @ce_option(True)
    def setWrap(self, value):
        option = self.document().defaultTextOption()
        if value:
            option.setWrapMode(option.WrapAtWordBoundaryOrAnywhere)
        else:
            option.setWrapMode(option.NoWrap)
        self.document().setDefaultTextOption(option)
# todo: move this bit to base class?
# This functionality embedded in the highlighter and even has a designated
# subpackage. I feel that it should be a part of the base editor.
# Note: if we do this, remove the hasattr call in the highlighter.
class SyntaxHighlighting(object):
    """Notes on syntax highlighting.
    The syntax highlighting/parsing is performed using three "components".
    The base component are the token instances. Each token simply represents
    a row of characters in the text the belong to each-other and should
    be styled in the same way. There is a token class for each particular
    "thing" in the code, such as comments, strings, keywords, etc. Some
    tokens are specific to a particular language.
    There is a function that produces a set of tokens, when given a line of
    text and a state parameter. There is such a function for each language.
    These "parsers" are defined in the parsers subpackage.
    And lastly, there is the Highlighter class, that applies the parser function
    to obtain the set of tokens and using the names of these tokens applies
    styling. The styling can be defined by giving a dict that maps token names
    to style representations.
    """
    # Register all syntax style elements
    _styleElements = Manager.getStyleElementDescriptionsForAllParsers()
    def parser(self):
        """parser()
        Get the parser instance currently in use to parse the code for
        syntax highlighting and source structure. Can be None.
        """
        try:
            return self.__parser
        except AttributeError:
            return None
    @ce_option(None)
    def setParser(self, parserName=""):
        """setParser(parserName='')
        Set the current parser by giving the parser name.
        """
        # Set parser
        self.__parser = Manager.getParserByName(parserName)
        # Restyle, use setStyle for lazy updating
        self.setStyle()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
"""
Code editor extensions that provides autocompleter functionality
"""
import pyzo
from ..qt import QtGui, QtCore, QtWidgets
Qt = QtCore.Qt
import keyword
# TODO: use this CompletionListModel to style the completion suggestions (class
names, method names, keywords etc)
class CompletionListModel(QtCore.QStringListModel):
    def data(self, index, role):
        if role == Qt.ForegroundRole:
            # data = str(QtWidgets.QStringListModel.data(self, index,
QtCore.Qt.DisplayRole))
            # return QtGui.QBrush(Qt.red)
            return None
        else:
            return QtCore.QStringListModel.data(self, index, role)
# todo: use keywords from the parser
class AutoCompletion(object):
    def __init__(self, *args, **kwds):
        super(AutoCompletion, self).__init__(*args, **kwds)
        # Autocompleter
        self.__completerModel = QtCore.QStringListModel(keyword.kwlist)
        self.__completer = QtWidgets.QCompleter(self)
        self.__completer.setModel(self.__completerModel)
        self.__completer.setCaseSensitivity(Qt.CaseInsensitive)
        self.__completer.setWidget(self)
        self.__completerNames = []
        self.__recentCompletions = []  # List of recently selected completions
        self.__cancelCallback = None
        # geometry
        self.__popupSize = 300, 100
        # Text position corresponding to first charcter of the word being
completed
        self.__autocompleteStart = None
        # We show the popup when this many chars have been input
        self.__autocompleteMinChars = 3
        self.__autocompleteFromObject = False
        self.__autocompleteVisible = False
        self.__autocompleteDebug = False
        self.__autocompletionAcceptKeys = (Qt.Key_Tab,)
        # Connect signals
        self.__highlightedCompletion = None
        self.__completer.activated.connect(self.onAutoComplete)
        self.__completer.highlighted.connect(self._setHighlightedCompletion)
    def _setHighlightedCompletion(self, value):
        """Keeping track of the highlighted item allows us
        to 'manually' perform an autocompletion.
        """
        self.__highlightedCompletion = value
    ## Properties
    def recentCompletionsList(self):
        """
        The list of recent auto-completions. This property may be set to a
        list that is shared among several editors, in order to share the notion
        of recent auto-completions
        """
        return self.__recentCompletions
    def setRecentCompletionsList(self, value):
        self.__recentCompletions = value
    def completer(self):
        return self.__completer
    def setAutoCompletionAcceptKeys(self, *keys):
        """Set the keys (Qt enums) that can accept an autocompletion.
        Like Tab, or Enter. Defaut Tab.
        """
        self.__autocompletionAcceptKeys = keys
    def setCancelCallback(self, cb):
        self.__cancelCallback = cb
    ## Autocompletion
    def setAutocompletPopupSize(self, width, height):
        """
        Set the size (width, heigth) of the automcompletion popup window.
        """
        self.__popupSize = width, height
    def setAutocompleteMinChars(self, n):
        """
        Set the number of chars where we show the popup.
        """
        self.__autocompleteMinChars = n
    def autocompleteShow(self, offset=0, names=None, fromObject=False):
        """
        Pop-up the autocompleter (if not already visible) and position it at
current
        cursor position minus offset. If names is given and not None, it is set
        as the list of possible completions.
        """
        # Pop-up the autocompleteList
        startcursor = self.textCursor()
        startcursor.movePosition(startcursor.Left, n=offset)
        self.__autocompleteFromObject = fromObject
        if self.__autocompleteDebug:
            print("autocompleteShow called")
        if names is not None:
            # TODO: a more intelligent implementation that adds new items and
removes
            # old ones
            if names != self.__completerNames:
                self.__completerModel.setStringList(names)
                self.__completerNames = names
        if (
            not self.autocompleteActive()
            or startcursor.position() != self.__autocompleteStart.position()
            or not self.autocompleteVisible()
        ):
            self.__autocompleteStart = startcursor
            self.__autocompleteStart.setKeepPositionOnInsert(True)
            # Popup the autocompleter. Don't use .complete() since we want to
            # position the popup manually
            self.__positionAutocompleter()
            if self.__updateAutocompleterPrefix():
                self.__autocompleteVisible = True
                self.__completer.popup().show()
            if self.__autocompleteDebug:
                print("self.__completer.popup().show() called")
        else:
            self.__updateAutocompleterPrefix()
    def autocompleteAccept(self):
        self.__completer.popup().hide()
        self.__autocompleteStart = None
        self.__autocompleteVisible = False
    def autocompleteCancel(self):
        self.__completer.popup().hide()
        self.__autocompleteStart = None
        self.__autocompleteVisible = False
        if self.__cancelCallback is not None:
            try:
                self.__cancelCallback()
            except Exception as err:
                print("Exception in autocomp cancel callback: " + str(err))
    def onAutoComplete(self, text=None):
        if text is None:
            text = self.__highlightedCompletion
        # Select the text from autocompleteStart until the current cursor
        cursor = self.textCursor()
        cursor.setPosition(self.__autocompleteStart.position(),
cursor.KeepAnchor)
        # Replace it with the selected text
        cursor.insertText(text)
        self.autocompleteAccept()  # Reset the completer
        # Update the recent completions list
        if text in self.__recentCompletions:
            self.__recentCompletions.remove(text)
        self.__recentCompletions.append(text)
    def autocompleteActive(self):
        """Returns whether an autocompletion list is currently started."""
        return self.__autocompleteStart is not None
    def autocompleteVisible(self):
        """Returns whether an autocompletion list is currently shown."""
        return self.__autocompleteVisible
    def __positionAutocompleter(self):
        """Move the autocompleter list to a proper position"""
        # Find the start of the autocompletion and move the completer popup
there
        cur = QtGui.QTextCursor(self.__autocompleteStart)  # Copy
__autocompleteStart
        # Set size
        geometry = self.__completer.popup().geometry()
        geometry.setWidth(self.__popupSize[0])
        geometry.setHeight(self.__popupSize[1])
        self.__completer.popup().setGeometry(geometry)
        # Initial choice for position of the completer
        position = self.cursorRect(cur).bottomLeft() + self.viewport().pos()
        # Check if the completer is going to go off the screen
        # top_geometry = QtWidgets.qApp.qApp.screens()[0].availableVirtualSize()
        top_geometry = pyzo.main.geometry()
        global_position = self.mapToGlobal(position)
        if global_position.y() + geometry.height() > top_geometry.height():
            # Move the completer to above the current line
            position = self.cursorRect(cur).topLeft() + self.viewport().pos()
            global_position = self.mapToGlobal(position)
            global_position -= QtCore.QPoint(0, int(geometry.height()))
        self.__completer.popup().move(global_position)
    def __updateAutocompleterPrefix(self):
        """
        Find the autocompletion prefix (the part of the word that has been
        entered) and send it to the completer. Update the selected completion
        (out of several possiblilties) which is best suited
        """
        if not self.autocompleteActive():
            self.__completer.popup().hide()  # TODO: why is this required?
            self.__autocompleteVisible = False
            return False
        # Select the text from autocompleteStart until the current cursor
        cursor = self.textCursor()
        cursor.setPosition(self.__autocompleteStart.position(),
cursor.KeepAnchor)
        prefix = cursor.selectedText()
        if (
            not self.__autocompleteFromObject
            and len(prefix) < self.__autocompleteMinChars
        ):
            self.__completer.setCompletionPrefix("")
            self.autocompleteCancel()
            return False
        else:
            self.__completer.setCompletionPrefix(prefix)
            model = self.__completer.completionModel()
            if model.rowCount():
                # Create a list of all possible completions, and select the one
                # which is best suited. Use the one which is highest in the
                # __recentCompletions list, but prefer completions with matching
                # case if they exists
                # Create a list of (row, value) tuples of all possible
completions
                completions = [
                    (
                        row,
                        model.data(
                            model.index(row, 0),
self.__completer.completionRole()
                        ),
                    )
                    for row in range(model.rowCount())
                ]
                # Define a function to get the position in the
__recentCompletions
                def completionIndex(data):
                    try:
                        return self.__recentCompletions.index(data)
                    except ValueError:
                        return -1
                # Sort twice; the last sort has priority over the first
                # Sort on most recent completions
                completions.sort(key=lambda c: completionIndex(c[1]),
reverse=True)
                # Sort on matching case (prefer matching case)
                completions.sort(key=lambda c: c[1].startswith(prefix),
reverse=True)
                # apply the best match
                bestMatchRow = completions[0][0]
self.__completer.popup().setCurrentIndex(model.index(bestMatchRow, 0))
                return True
            else:
                # No match, just hide
                self.autocompleteCancel()
                return False
    def potentiallyAutoComplete(self, event):
        """potentiallyAutoComplete(event)
        Given a keyEvent, check if we should perform an autocompletion.
        Returns 0 if no autocompletion was performed. Return 1 if
        autocompletion was performed, but the key event should be processed
        as normal. Return 2 if the autocompletion was performed, and the key
        should be consumed.
        """
        if self.autocompleteActive():
            if event.key() in self.__autocompletionAcceptKeys:
                if event.key() <= 128:
                    self.onAutoComplete()  # No arg: select last highlighted
                    event.ignore()
                    return 1  # Let key have effect as normal
                elif event.modifiers() == Qt.NoModifier:
                    # The key
                    self.onAutoComplete()  # No arg: select last highlighted
                    return 2  # Key should be consumed
        return 0
    def keyPressEvent(self, event):
        key = event.key()
        modifiers = event.modifiers()
        if (
            key == Qt.Key_Escape
            and modifiers == Qt.NoModifier
            and self.autocompleteActive()
        ):
            self.autocompleteCancel()
            return  # Consume the key
        if self.potentiallyAutoComplete(event) > 1:
            return  # Consume
        # Allowed keys that do not close the autocompleteList:
        # alphanumeric and _ ans shift
        # Backspace (until start of autocomplete word)
        if (
            self.autocompleteActive()
            and not event.text().isalnum()
            and event.text() != "_"
            and key != Qt.Key_Shift
            and not (
                (key == Qt.Key_Backspace)
                and self.textCursor().position() >
self.__autocompleteStart.position()
            )
        ):
            self.autocompleteCancel()
        # Apply the key that was pressed
        super(AutoCompletion, self).keyPressEvent(event)
        if self.autocompleteActive():
            # While we type, the start of the autocompletion may move due to
line
            # wrapping, so reposition after every key stroke
            self.__positionAutocompleter()
            self.__updateAutocompleterPrefix()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
"""
Code editor extensions that change its behaviour (i.e. how it reacts to keys)
"""
from ..qt import QtGui, QtCore
Qt = QtCore.Qt
import pyzo
from ..misc import ustr, ce_option
from ..parsers.tokens import (
    CommentToken,
    IdentifierToken,
    StringToken,
    UnterminatedStringToken,
)
from ..parsers.python_parser import MultilineStringToken
from ..parsers import BlockState
class MoveLinesUpDown(object):
    def keyPressEvent(self, event):
        if event.key() in (Qt.Key_Up, Qt.Key_Down) and (
            Qt.ControlModifier & event.modifiers()
            and Qt.ShiftModifier & event.modifiers()
        ):
            cursor = self.textCursor()
            cursor.beginEditBlock()
            try:
                self._swaplines(cursor, event.key())
            finally:
                cursor.endEditBlock()
        else:
            super().keyPressEvent(event)
    def _swaplines(self, cursor, key):
        # Get positions of selection
        start = cursor.selectionStart()
        end = cursor.selectionEnd()
        # Get text of selected blocks
        cursor.setPosition(start, cursor.MoveAnchor)
        cursor.movePosition(cursor.StartOfBlock, cursor.MoveAnchor)
        cursor.setPosition(end, cursor.KeepAnchor)
        cursor.movePosition(cursor.EndOfBlock, cursor.KeepAnchor)
        text1 = cursor.selectedText()
        cursor.removeSelectedText()
        pos1 = cursor.position()
        # Move up/down
        other = [cursor.NextBlock, cursor.PreviousBlock][int(bool(key ==
Qt.Key_Up))]
        cursor.movePosition(other, cursor.MoveAnchor)
        # Select text of other block
        cursor.movePosition(cursor.StartOfBlock, cursor.MoveAnchor)
        cursor.movePosition(cursor.EndOfBlock, cursor.KeepAnchor)
        text2 = cursor.selectedText()
        cursor.removeSelectedText()
        pos2 = cursor.position()
        # Insert text
        cursor.insertText(text1)
        pos3 = cursor.position()
        # Move back
        if key == Qt.Key_Up:
            cursor.movePosition(cursor.NextBlock, cursor.MoveAnchor)
        else:
            cursor.setPosition(pos1, cursor.MoveAnchor)
            pos2 += len(text2)
            pos3 += len(text2)
        # Replace text
        cursor.insertText(text2)
        # Leave original lines selected for continued movement
        cursor.setPosition(pos2, cursor.MoveAnchor)
        cursor.setPosition(pos3, cursor.KeepAnchor)
        self.setTextCursor(cursor)
class ScrollWithUpDownKeys(object):
    def keyPressEvent(self, event):
        if (
            event.key() in (Qt.Key_Up, Qt.Key_Down)
            and Qt.ControlModifier == event.modifiers()
        ):
            s = self.verticalScrollBar()
            # h = self.cursorRect(self.textCursor()).height()
            if event.key() == Qt.Key_Up:
                s.setValue(s.value() + 1)
            else:
                s.setValue(s.value() - 1)
        else:
            super().keyPressEvent(event)
```

```
class HomeKey(object):
    def keyPressEvent(self, event):
        # Home or shift + home
        if event.key() == Qt.Key_Home and event.modifiers() in (
            Qt.NoModifier,
            Qt.ShiftModifier,
        ):
            # Prepare
            cursor = self.textCursor()
            shiftDown = event.modifiers() == Qt.ShiftModifier
            moveMode = [cursor.MoveAnchor, cursor.KeepAnchor][shiftDown]
            # Get leading whitespace
            text = ustr(cursor.block().text())
            leadingWhitespace = text[: len(text) - len(text.lstrip())]
            # Get current position and move to start of whitespace
            i = cursor.positionInBlock()
            cursor.movePosition(cursor.StartOfBlock, moveMode)
            cursor.movePosition(cursor.Right, moveMode, len(leadingWhitespace))
            # If we were alread there, move to start of block
            if cursor.positionInBlock() == i:
                cursor.movePosition(cursor.StartOfBlock, moveMode)
            # Done
            self.setTextCursor(cursor)
        else:
            super().keyPressEvent(event)
class EndKey(object):
    def keyPressEvent(self, event):
        if event.key() == Qt.Key_End and event.modifiers() in (
            Qt.NoModifier,
            Qt.ShiftModifier,
        ):
            # Prepare
            cursor = self.textCursor()
            shiftDown = event.modifiers() == Qt.ShiftModifier
            moveMode = [cursor.MoveAnchor, cursor.KeepAnchor][shiftDown]
            # Get current position and move to end of line
            i = cursor.positionInBlock()
            cursor.movePosition(cursor.EndOfLine, moveMode)
            # If alread at end of line, move to end of block
            if cursor.positionInBlock() == i:
                cursor.movePosition(cursor.EndOfBlock, moveMode)
            # Done
            self.setTextCursor(cursor)
        else:
            super().keyPressEvent(event)
class NumpadPeriodKey(object):
    """
    If the numpad decimal separator key is pressed, always insert
    a period (.) even if due to localization that key is mapped to a
    comma (,). When editing code, period is the decimal separator
    independent of localization
    """
    # NOTE: THIS PLUGIN IS DISABLED, see issue #720
    def keyPressEvent(self, event):
        # Check for numpad comma
        if (
            event.key() == QtCore.Qt.Key_Comma
            and event.modifiers() & QtCore.Qt.KeypadModifier
        ):
            # Create a new QKeyEvent to substitute the original one
            event = QtGui.QKeyEvent(
                event.type(),
                QtCore.Qt.Key_Period,
                event.modifiers(),
                ".",
                event.isAutoRepeat(),
                event.count(),
            )
        super().keyPressEvent(event)
class Indentation(object):
    def __cursorIsInLeadingWhitespace(self, cursor=None):
        """
        Checks wether the given cursor is in the leading whitespace of a block,
i.e.
        before the first non-whitespace character. The cursor is not modified.
        If the cursor is not given or is None, the current textCursor is used
        """
        if cursor is None:
            cursor = self.textCursor()
        # Get the text of the current block up to the cursor
        textBeforeCursor = ustr(cursor.block().text())[:
cursor.positionInBlock()]
        return (
            textBeforeCursor.lstrip() == ""
        )  # If we trim it and it is empty, it's all whitespace
    def keyPressEvent(self, event):
        key = event.key()
        modifiers = event.modifiers()
        # Tab key
        if key == Qt.Key_Tab:
            if modifiers == Qt.NoModifier:
                if (
                    self.textCursor().hasSelection()
                ):  # Tab pressed while some area was selected
                    self.indentSelection()
                    return
                elif self.__cursorIsInLeadingWhitespace():
                    # If the cursor is in the leading whitespace, indent and
move cursor to end of whitespace
                    cursor = self.textCursor()
                    self.indentBlock(cursor)
                    self.setTextCursor(cursor)
                    return
                elif self.indentUsingSpaces():
                    # Insert space-tabs
                    cursor = self.textCursor()
                    w = self.indentWidth()
                    cursor.insertText(" " * (w - ((cursor.positionInBlock() + w)
% w)))
                    return
                # else: default behaviour, insert tab character
            else:  # Some other modifiers + Tab: ignore
                return
        # If backspace is pressed in the leading whitespace, (except for at the
first
        # position of the line), and there is no selection
        # dedent that line and move cursor to end of whitespace
        if (
            key == Qt.Key_Backspace
            and modifiers == Qt.NoModifier
            and self.__cursorIsInLeadingWhitespace()
            and not self.textCursor().atBlockStart()
            and not self.textCursor().hasSelection()
        ):
            # Create a cursor, dedent the block and move screen cursor to the
end of the whitespace
            cursor = self.textCursor()
            self.dedentBlock(cursor)
            self.setTextCursor(cursor)
            return
        # todo: Same for delete, I think not (what to do with the cursor?)
        # Auto-unindent
        if key == Qt.Key_Delete:
            cursor = self.textCursor()
            if not cursor.hasSelection():
                cursor.movePosition(cursor.EndOfBlock, cursor.KeepAnchor)
                if not cursor.hasSelection() and
cursor.block().next().isValid():
                    cursor.beginEditBlock()
                    cursor.movePosition(cursor.NextBlock)
                    self.indentBlock(cursor, -99)  # dedent as much as we can
                    cursor.deletePreviousChar()
                    cursor.endEditBlock()
                    return
        super().keyPressEvent(event)
class AutoIndent(object):
    """
    Auto indentation. This extension only adds the autoIndent property, for the
    actual indentation, the editor should derive from some AutoIndenter object
    """
    def autoIndent(self):
        """autoIndent()
        Get whether auto indentation is enabled.
        """
        return self.__autoIndent
    @ce_option(True)
    def setAutoIndent(self, value):
        """setAutoIndent(value)
        Set whether to enable auto indentation.
        """
        self.__autoIndent = bool(value)
class PythonAutoIndent(object):
    def keyPressEvent(self, event):
        super().keyPressEvent(event)
        if not self.autoIndent():
            return
        # This extension code is run *after* key is processed by QPlainTextEdit
        if event.key() in (Qt.Key_Enter, Qt.Key_Return):
            cursor = self.textCursor()
            # Prevent in-block newlines (issue #482)
            if not cursor.atBlockStart() and not cursor.hasSelection():
                cursor.deletePreviousChar()
                cursor.insertBlock()
                cursor = self.textCursor()
            previousBlock = cursor.block().previous()
            if previousBlock.isValid():
                line = ustr(previousBlock.text())
                indent = line[: len(line) - len(line.lstrip())]
                if line.endswith(":"):
                    # We only need to add indent if the : is not in a
(multiline)
                    # string or comment. Therefore, find out what the syntax
                    # highlighter thinks of the previous line.
                    ppreviousBlock = (
                        previousBlock.previous()
                    )  # the block before previous
                    ppreviousState = (
                        ppreviousBlock.userState() if previousBlock.isValid()
else 0
                    )
                    lastElementToken = list(
                        self.parser().parseLine(previousBlock.text(),
ppreviousState)
                    )[-1]
                    # Because there's at least a : on that line, the list is
never empty
                    if not isinstance(
                        lastElementToken,
                        (CommentToken, UnterminatedStringToken, BlockState),
                    ):
                        # TODO: check correct identation (no mixed space/tabs)
                        if self.indentUsingSpaces():
                            indent += " " * self.indentWidth()
                        else:
                            indent += "\t"
                cursor.insertText(indent)
                # This prevents jump to start of line when up key is pressed
                self.setTextCursor(cursor)
class SmartCopyAndPaste(object):
    """
    Smart copy and paste allows copying and pasting blocks
    """
    @staticmethod
    def __setCursorPositionAndAnchor(cursor, position, anchor):
        cursor.setPosition(anchor)
        cursor.setPosition(position, cursor.KeepAnchor)
    @classmethod
    def __ensureCursorBeforeAnchor(cls, cursor):
        """
        Given a cursor, modify it such that the cursor.position() is before or
        at cursor.anchor() and not the other way around.
        Returns: anchorBeforeCursor, i.e. whether originally the anchor was
        before the cursor
        """
        start = cursor.selectionStart()
        end = cursor.selectionEnd()
        # Remember whether the cursor is before or after the anchor
        anchorBeforeCursor = cursor.anchor() < cursor.position()
        cls.__setCursorPositionAndAnchor(cursor, start, end)
        # Return wheter originally the cursor was before the anchor
        return anchorBeforeCursor
    def copy(self):
        """
        Smart copy: if selection is multi-line and in front of the start of the
        selection there is only whitespace, extend the selection to include only
        whitespace
        """
        cursor = self.textCursor()
        start = cursor.selectionStart()
        end = cursor.selectionEnd()
        # For our convenience, ensure that position is at start and
        # anchor is at the end, but remember whether originally the
        # anchor was before the cursor or the other way around
        anchorBeforeCursor = self.__ensureCursorBeforeAnchor(cursor)
        # Check if we have multi-line selection.
        block = cursor.block()
        # Use > not >= to ensure we don't count it as multi-line if the cursor
        # is just at the beginning of the next block (consistent with
'CodeEditor.doForSelectedLines')
        if end > (block.position() + block.length()):
            # Now check if there is only whitespace before the start of
selection
            # If so, include this whitespace in the selection and update the
            # selection of the editor
            textBeforeSelection = block.text()[: cursor.positionInBlock()]
            if len(textBeforeSelection.strip()) == 0:
                start = block.position()  # Move start to include leading
whitespace
```

```
                # Update the textcursor of our editor. If originally the
                # anchor was before the cursor, restore that situation
                if anchorBeforeCursor:
                    self.__setCursorPositionAndAnchor(cursor, end, start)
                else:
                    self.__setCursorPositionAndAnchor(cursor, start, end)
                self.setTextCursor(cursor)
        # Call our supers copy slot to do the actual copying
        super().copy()
    def cut(self):
        """
        Cutting with smart-copy: the part that is copies is the same as
self.copy(),
        but the part that is removed is only the original selection
        see: Qt qtextcontrol.cpp, cut()
        """
        if (
            self.textInteractionFlags() & QtCore.Qt.TextEditable
        ) and self.textCursor().hasSelection():
            cursor = self.textCursor()
            self.copy()
            # Restore original cursor
            self.setTextCursor(cursor)
            cursor.removeSelectedText()
    def paste(self):
        """
        Smart paste
        If you paste on a position that has only whitespace in front of it,
        remove the whitespace before pasting. Combined with smart copy,
        this ensure indentation of the
        """
        self._paste(keepSelection=False)
    def pasteAndSelect(self):
        """
        Smart paste
        Like paste(), but keep the part that was pasted selected. This allows
        you to change the indentation after pasting using tab / shift-tab
        """
        self._paste(keepSelection=True)
    def _paste(self, keepSelection):
        # Create a cursor of the current selection
        cursor = self.textCursor()
        # Ensure that position is at start and anchor is at the end.
        # This is required to ensure that with keepPositiobOnInsert
        # set, the cursor will equal the pasted text after the pasting
        self.__ensureCursorBeforeAnchor(cursor)
        # Change this cursor to let the position() stay at its place upon
        # inserting the new code; the anchor will move to the end of the
insertion
        cursor.setKeepPositionOnInsert(True)
        super().paste()
        block = cursor.block()
        # Check if the thing to be pasted is multi-line. Use > not >=
        # to ensure we don't count it as multi-line if the cursor
        # is just at the beginning of the next block (consistent with
        # 'CodeEditor.doForSelectedLines')
        if cursor.selectionEnd() > block.position() + block.length():
            # Now, check if in front of the current selection there is only
whitespace
            if len(block.text()[: cursor.positionInBlock()].strip()) == 0:
                # Note that this 'smart pasting' will be a separate item on the
                # undo stack. This is intentional: the user can undo the 'smart
paste'
                # without undoing the paste
                cursor2 = QtGui.QTextCursor(cursor)
                cursor2.setPosition(
                    cursor2.position()
                )  # put the anchor where the cursor is
                # Move cursor2 to beginning of the line (selecting the
whitespace)
                # and remove the selection
                cursor2.movePosition(cursor2.StartOfBlock, cursor2.KeepAnchor)
                cursor2.removeSelectedText()
        # Set the textcursor of this editor to the just-pasted selection
        if keepSelection:
            cursor.setKeepPositionOnInsert(False)
            self.setTextCursor(cursor)
class AutoCloseQuotesAndBrackets(object):
    """
    Automatic insertion of quotes, parenthesis, braces and brackets
    """
    def _get_token_at_cursor(self, cursor=None, relpos=0):
        """Get token at the (current or given) cursor position. Can be None."""
        if cursor is None:
            cursor = self.textCursor()
        pos = cursor.positionInBlock() + relpos
        tokens = cursor.block().userData().tokens
        token = None
        for token in tokens:
            if hasattr(token, "start"):
                if token.start >= pos:
                    break
            elif getattr(token, "state", 0) in (1, 2):
                token = MultilineStringToken()  # 1 and 2 are mls, by
convention, sortof
        return token
    def keyPressEvent(self, event):
        try:
            self.__keyPressEvent(event)
        except Exception as err:
            # When there is a bug in our fancy autoclosing stuff, better print
and
            # and have the plain behavior instead of not working ...
            print(err)
            super().keyPressEvent(event)
    def __keyPressEvent(self, event):
        quotes = "'", '"'
        openBrackets = "{", "[", "("
        closeBrackets = "}", "]", ")"
        brackets = openBrackets + closeBrackets
        cursor = self.textCursor()
        char = event.text()
        #  brackets
        if char in brackets and pyzo.config.settings.autoClose_Brackets:
            # Dont autobracket inside comments and strings
            if isinstance(
                self._get_token_at_cursor(cursor),
                (
                    CommentToken,
                    StringToken,
                    MultilineStringToken,
                    UnterminatedStringToken,
                ),
            ):
                super().keyPressEvent(event)
                return
            if char in openBrackets:
                idx = openBrackets.index(char)
                next_char = self.__getNextCharacter()
                if cursor.selectedText():
                    # Surround selection with brackets
                    new_text = (
                        openBrackets[idx] + cursor.selectedText() +
closeBrackets[idx]
                    )
                    cursor.setKeepPositionOnInsert(True)
                    cursor.insertText(new_text)
                    cursor.setKeepPositionOnInsert(False)
                    self.setTextCursor(cursor)
                elif (
                    next_char.strip() and next_char not in closeBrackets
                ):  # str.strip() conveniently removes all kinds of whitespace
                    # Only autoclose if the char on the right is whitespace
                    cursor.insertText(char)  # == super().keyPressEvent(event)
                else:
                    # Auto-close bracket
                    insert_txt = "{}{}".format(openBrackets[idx],
closeBrackets[idx])
                    cursor.insertText(insert_txt)
                    self._moveCursorLeft(1)
            elif char in closeBrackets:
                idx = closeBrackets.index(char)
                next_char = self.__getNextCharacter()
                if cursor.selectedText():
                    # Replace
                    cursor.insertText(char)
                elif next_char == char:
                    # Skip
                    self._moveCursorRight(1)
                else:
                    # Normal
                    cursor.insertText(char)  # == super().keyPressEvent(event)
        # quotes
        elif char in quotes and pyzo.config.settings.autoClose_Quotes:
            next_char = self.__getNextCharacter()
            # Dont autoquote inside comments and multiline strings
            # Only allow "doing our thing" when we're at the end of a normal
string
            token = self._get_token_at_cursor(cursor)
            if isinstance(token, (CommentToken, MultilineStringToken)):
                super().keyPressEvent(event)
                return
            elif isinstance(token, StringToken) and char != next_char:
                super().keyPressEvent(event)
                return
            if cursor.selectedText() in ('"', "'"):
                # Skip over char if its one char and a quote
                self._moveCursorRight(1)
            elif cursor.selectedText():
                # Surround selection with quotes, maybe even multi-line
                new_text = char + cursor.selectedText() + char
                if "\u2029" in new_text and "python" in 
self.parser().name().lower():
                    new_text = char * 2 + new_text + char * 2
                cursor.setKeepPositionOnInsert(True)
                cursor.insertText(new_text)
                cursor.setKeepPositionOnInsert(False)
                self.setTextCursor(cursor)
            elif next_char == char:
                # Skip
                self._moveCursorRight(1)
            else:
                # Only autoquote if we're next to whitespace, operator, quote
                notok_token_types = (IdentifierToken,)
                tokenL = self._get_token_at_cursor(cursor, -1)
                tokenR = self._get_token_at_cursor(cursor, -0)
                if isinstance(tokenL, notok_token_types) or isinstance(
                    tokenR, notok_token_types
                ):
                    super().keyPressEvent(event)
                    return
                # Auto-close
                cursor.insertText(char * 2)
                self._moveCursorLeft(1)
                # # Maybe handle tripple quotes (add 2 more if we now have 3 on 
the left)
                # Disabled: this feature too easily gets in the way
                # if 'python' in self.parser().name().lower():
                #     cursor = self.textCursor()
                #     cursor.movePosition(cursor.PreviousCharacter, 
cursor.KeepAnchor, 3)
                #     if cursor.selectedText() == char * 3:
                #         edit_cursor = self.textCursor()
                #         edit_cursor.insertText(char * 2)
                #         self._moveCursorLeft(2)
```

```
        # remove whole couple of brackets when hitting backspace
        elif (
            event.key() == Qt.Key_Backspace and
pyzo.config.settings.autoClose_Brackets
        ):
            if isinstance(
                self._get_token_at_cursor(cursor),
                (
                    CommentToken,
                    StringToken,
                    MultilineStringToken,
                    UnterminatedStringToken,
                ),
            ):
                super().keyPressEvent(event)
                return
            else:
                prev_char = self.__getPreviousCharacter()
                next_char = self.__getNextCharacter()
                if prev_char == "(" and next_char == ")":
                    cursor.deleteChar()
                elif prev_char == "[" and next_char == "]":
                    cursor.deleteChar()
                elif prev_char == "{" and next_char == "}":
                    cursor.deleteChar()
                super().keyPressEvent(event)
        else:
            super().keyPressEvent(event)
    def __getNextCharacter(self):
        cursor = self.textCursor()
        cursor.movePosition(cursor.NoMove, cursor.MoveAnchor)  # rid selection
        cursor.movePosition(cursor.NextCharacter, cursor.KeepAnchor)
        next_char = cursor.selectedText()
        return next_char
    def __getPreviousCharacter(self):
        cursor = self.textCursor()
        cursor.movePosition(cursor.NoMove, cursor.MoveAnchor)  # rid selection
        cursor.movePosition(cursor.PreviousCharacter, cursor.KeepAnchor)
        previous_char = cursor.selectedText()
        return previous_char
    def _moveCursorLeft(self, n):
        """
        Move cursor left between eg. brackets
        """
        cursor2 = self.textCursor()
        cursor2.movePosition(cursor2.Left, cursor2.MoveAnchor, n)
        self.setTextCursor(cursor2)
    def _moveCursorRight(self, n):
        """
        Move cursor out of eg. brackets
        """
        cursor2 = self.textCursor()
        cursor2.movePosition(cursor2.Right, cursor2.MoveAnchor, n)
        self.setTextCursor(cursor2)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
from ..qt import QtCore, QtGui, QtWidgets  # noqa
Qt = QtCore.Qt
class Calltip(object):
    _styleElements = [
        (
            "Editor.calltip",
            "The style of the calltip. ",
            "fore:#555, back:#ff9, border:1",
        )
    ]
    class __CalltipLabel(QtWidgets.QLabel):
        def __init__(self):
            QtWidgets.QLabel.__init__(self)
            # Start hidden
            self.hide()
            # Accept rich text
            self.setTextFormat(QtCore.Qt.RichText)
            # Show as tooltip
            self.setIndent(2)
            self.setWindowFlags(QtCore.Qt.ToolTip)
        def enterEvent(self, event):
            # Act a bit like a tooltip
            self.hide()
    def __init__(self, *args, **kwds):
        super(Calltip, self).__init__(*args, **kwds)
        # Create label for call tips
        self.__calltipLabel = self.__CalltipLabel()
        # Be notified of style updates
        self.styleChanged.connect(self.__afterSetStyle)
        # Prevents calltips from being shown immediately after pressing
        # the escape key.
        self.__noshow = False
    def __afterSetStyle(self):
        format = self.getStyleElementFormat("editor.calltip")
        ss = "QLabel { color:%s; background:%s; border:%ipx solid %s; }" % (
            format["fore"],
            format["back"],
            int(format["border"]),
            format["fore"],
        )
        self.__calltipLabel.setStyleSheet(ss)
    def calltipShow(self, offset=0, richText="", highlightFunctionName=False):
        """calltipShow(offset=0, richText='', highlightFunctionName=False)
        Shows the given calltip.
        Parameters
        ----------
        offset : int
            The character offset to show the tooltip.
        richText : str
            The text to show (may contain basic html for markup).
        highlightFunctionName : bool
            If True the text before the first opening brace is made bold.
            default False.
        """
        # Do not show the calltip if it was deliberately hidden by the
        # user.
        if self.__noshow:
            return
        # Process calltip text?
        if highlightFunctionName:
            i = richText.find("(")
            if i > 0:
                richText = "<b>{}</b>{}".format(richText[:i], richText[i:])
        # Get a cursor to establish the position to show the calltip
        startcursor = self.textCursor()
        startcursor.movePosition(startcursor.Left, n=offset)
        # Get position in pixel coordinates
        rect = self.cursorRect(startcursor)
        pos = rect.topLeft()
        pos.setY(pos.y() - rect.height() - 1)  # Move one above line
        pos.setX(pos.x() - 3)  # Correct for border and indent
        pos = self.viewport().mapToGlobal(pos)
        # Set text and update font
        self.__calltipLabel.setText(richText)
        self.__calltipLabel.setFont(self.font())
        # Use a qt tooltip to show the calltip
        if richText:
            self.__calltipLabel.move(pos)
            self.__calltipLabel.show()
        else:
            self.__calltipLabel.hide()
    def calltipCancel(self):
        """calltipCancel()
        Hides the calltip.
        """
        self.__calltipLabel.hide()
    def calltipActive(self):
        """calltipActive()
        Get whether the calltip is currently active.
        """
        return self.__calltipLabel.isVisible()
    def focusOutEvent(self, event):
        super(Calltip, self).focusOutEvent(event)
        self.__calltipLabel.hide()
    def keyPressEvent(self, event):
        # If the user presses Escape and the calltip is active, hide it
        if (
            event.key() == Qt.Key_Escape
            and event.modifiers() == Qt.NoModifier
            and self.calltipActive()
        ):
            self.calltipCancel()
            self.__noshow = True
            return
        if event.text() == "(":
            self.__noshow = False
        elif event.text() == ")":
            self.calltipCancel()
        # Proceed processing the keystrike
        super(Calltip, self).keyPressEvent(event)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
# Import tokens in module namespace
from .python_parser import PythonParser, pythonKeywords
# Set keywords
cythonExtraKeywords = set(
    ["cdef", "cpdef", "ctypedef", "cimport", "float", "double", "int", "long"]
)
class CythonParser(PythonParser):
    """Parser for Cython/Pyrex."""
    _extensions = ["pyi", ".pyx", ".pxd"]
    _keywords = pythonKeywords | cythonExtraKeywords
    def _identifierState(self, identifier=None):
        """Given an identifier returs the identifier state:
        3 means the current identifier can be a function.
        4 means the current identifier can be a class.
        0 otherwise.
        This method enables storing the state during the line,
        and helps the Cython parser to reuse the Python parser's code.
        This implementation keeps a counter. If the counter is 0, the
        state is zero.
        """
        if identifier is None:
            # Explicit get and reset
            state = 0
            try:
                if self._idsCounter > 0:
                    state = self._idsState
            except Exception:
                pass
            self._idsState = 0
            self._idsCounter = 0
            return state
        elif identifier in ["def", "cdef", "cpdef"]:
            # Set function state
            self._idsState = 3
            self._idsCounter = 2
            return 3
        elif identifier == "class":
            # Set class state
```

```
                self._idsState = 4
                self._idsCounter = 1
                return 4
            elif self._idsCounter > 0:
                self._idsCounter -= 1
                return self._idsState
            else:
                # This one can be func or class, next one can't
                return 0
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
import re
from . import Parser, BlockState, text_type
from .tokens import ALPHANUM
from .tokens import (
    Token,
    CommentToken,
    StringToken,
    UnterminatedStringToken,
    IdentifierToken,
    NonIdentifierToken,
    KeywordToken,
    NumberToken,
)
# todo: compiler directives (or how do you call these things starting with #)
class MultilineCommentToken(CommentToken):
    """Characters representing a multi-line comment."""
    defaultStyle = "fore:#007F00"
class CharToken(Token):
    """Single-quoted char"""
    defaultStyle = "fore:#7F007F"
# This regexp is used to find special stuff, such as comments, numbers and
# strings.
tokenProg = re.compile(
    "(["
    + ALPHANUM
    + "_]+)|"
    + "(\/\/)|"  # Identifiers/numbers (group 1) or
    + "(\/\*)|"  # Single line comment (group 2)
    + "('\\\\?.')|"  # Comment (group 3) or
    + '(")'  # char (group 4)  # string (group 5)
)
# For a string, get the RegExp
# program that matches the end. (^|[^\\]) means: start of the line
# or something that is not \ (since \ is supposed to escape the following
# quote) (\\\\)* means: any number of two slashes \\ since each slash will
# escape the next one
stringEndProg = re.compile(r'(^|[^\\])(\\\\)*"')
commentEndProg = re.compile(r"\*/")
```

```
class CParser(Parser):
    """A C parser."""
    _extensions = [".c", ".h", ".cpp", "cxx", "hxx"]
    _keywords = ["int", "const", "char", "void", "short", "long", "case"]
    def parseLine(self, line, previousState=0):
        """parseLine(line, previousState=0)
        Parses a line of C code, yielding tokens.
        """
        line = text_type(line)
        pos = 0  # Position following the previous match
        # identifierState and previousstate values:
        # 0: nothing special
        # 1: string
        # 2: multiline comment /* */
        # First determine whether we should look for the end of a string,
        # or if we should process a token.
        if previousState == 1:
            token = StringToken(line, 0, 0)
            tokens = self._findEndOfString(line, token)
            # Process tokens
            for token in tokens:
                yield token
                if isinstance(token, BlockState):
                    return
            pos = token.end
        elif previousState == 2:
            token = MultilineCommentToken(line, 0, 0)
            tokens = self._findEndOfComment(line, token)
            # Process tokens
            for token in tokens:
                yield token
                if isinstance(token, BlockState):
                    return
            pos = token.end
        # Enter the main loop that iterates over the tokens and skips strings
        while True:
            # Get next tokens
            tokens = self._findNextToken(line, pos)
            if not tokens:
                return
            elif isinstance(tokens[-1], StringToken):
                moreTokens = self._findEndOfString(line, tokens[-1])
                tokens = tokens[:-1] + moreTokens
            elif isinstance(tokens[-1], MultilineCommentToken):
                moreTokens = self._findEndOfComment(line, tokens[-1])
                tokens = tokens[:-1] + moreTokens
            # Process tokens
            for token in tokens:
                yield token
                if isinstance(token, BlockState):
                    return
            pos = token.end
    def _findEndOfComment(self, line, token):
        """Find the matching comment end in the rest of the line"""
        # Do not use the start parameter of search, since ^ does not work then
        endMatch = commentEndProg.search(line, token.end)
        if endMatch:
            # The comment does end on this line
            token.end = endMatch.end()
            return [token]
        else:
            # The comment does not end on this line
            token.end = len(line)
            return [token, BlockState(2)]
    def _findEndOfString(self, line, token):
        """Find the matching string end in the rest of the line"""
        # todo: distinguish between single and double quote strings
        # Find the matching end in the rest of the line
        # Do not use the start parameter of search, since ^ does not work then
        endMatch = stringEndProg.search(line[token.end :])
        if endMatch:
            # The string does end on this line
            token.end = token.end + endMatch.end()
            return [token]
        else:
            # The string does not end on this line
            if line.strip().endswith("\\"):  # Multi line string
                token = StringToken(line, token.start, len(line))
                return [token, BlockState(1)]
            else:
                return [UnterminatedStringToken(line, token.start, len(line))]
    def _findNextToken(self, line, pos):
        """_findNextToken(line, pos):
        Returns a token or None if no new tokens can be found.
        """
        # Init tokens, if positing too large, stop now
        if pos > len(line):
            return None
        tokens = []
        # Find the start of the next string or comment
        match = tokenProg.search(line, pos)
        # Process the Non-Identifier between pos and match.start()
        # or end of line
        nonIdentifierEnd = match.start() if match else len(line)
        # Return the Non-Identifier token if non-null
        token = NonIdentifierToken(line, pos, nonIdentifierEnd)
        if token:
            tokens.append(token)
        # If no match, we are done processing the line
        if not match:
            return tokens
        # The rest is to establish what identifier we are dealing with
        # Identifier ("a word or number") Find out whether it is a key word
        if match.group(1) is not None:
            identifier = match.group(1)
            tokenArgs = line, match.start(), match.end()
            if identifier in self._keywords:
                tokens.append(KeywordToken(*tokenArgs))
            elif identifier[0] in "0123456789":
                # identifierState = 0
                tokens.append(NumberToken(*tokenArgs))
            else:
                tokens.append(IdentifierToken(*tokenArgs))
        # Single line comment
        elif match.group(2) is not None:
            tokens.append(CommentToken(line, match.start(), len(line)))
        elif match.group(3) is not None:
            tokens.append(MultilineCommentToken(line, match.start(),
match.end()))
        elif match.group(4) is not None:  # Char
            tokens.append(CharToken(line, match.start(), match.end()))
        else:
            # We have matched a string-start
            tokens.append(StringToken(line, match.start(), match.end()))
        # Done
        return tokens
if __name__ == "__main__":
    parser = CParser()
    for token in parser.parseLine("void test(int i=2) /* test "):
        print("%s %s" % (token.name, token))
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
import re
from . import Parser, BlockState, text_type
from .tokens import ALPHANUM
from ..misc import ustr
# Import tokens in module namespace
from .tokens import (
    CommentToken,
    StringToken,
    UnterminatedStringToken,
    IdentifierToken,
    NonIdentifierToken,
    KeywordToken,
    BuiltinsToken,
    InstanceToken,
    NumberToken,
    FunctionNameToken,
    ClassNameToken,
    TodoCommentToken,
    OpenParenToken,
    CloseParenToken,
    IllegalToken,
)
# Keywords sets
# Source: import keyword; keyword.kwlist (Python 2.6.6)
python2Keywords = set(
    [
        "and",
        "as",
        "assert",
        "break",
        "class",
        "continue",
        "def",
        "del",
        "elif",
        "else",
        "except",
        "exec",
        "finally",
        "for",
        "from",
        "global",
        "if",
        "import",
        "in",
        "is",
        "lambda",
        "not",
        "or",
        "pass",
        "print",
        "raise",
        "return",
        "try",
        "while",
        "with",
        "yield",
    ]
)
# Source: import keyword; keyword.kwlist (Python 3.1.2)
python3Keywords = set(
    [
        "False",
        "None",
        "True",
        "and",
        "as",
        "assert",
        "break",
        "class",
        "continue",
        "def",
        "del",
        "elif",
        "else",
        "except",
        "finally",
        "for",
        "from",
        "global",
        "if",
        "import",
        "in",
        "is",
        "lambda",
        "nonlocal",
        "not",
        "or",
        "pass",
        "raise",
        "return",
        "try",
        "while",
        "with",
        "yield",
        "async",
        "await",
    ]
)
# Merge the two sets to get a general Python keyword list
pythonKeywords = python2Keywords | python3Keywords
# Builtins sets
# Source: dir (__builtins__) (Python 2.7.12)
python2Builtins = set(
    [
        "ArithmeticError",
        "AssertionError",
        "AttributeError",
        "BaseException",
        "BufferError",
        "BytesWarning",
        "DeprecationWarning",
        "EOFError",
        "Ellipsis",
        "EnvironmentError",
        "Exception",
        "False",
        "FloatingPointError",
        "FutureWarning",
        "GeneratorExit",
        "IOError",
        "ImportError",
        "ImportWarning",
        "IndentationError",
        "IndexError",
        "KeyError",
        "KeyboardInterrupt",
        "LookupError",
        "MemoryError",
        "NameError",
        "None",
        "NotImplemented",
        "NotImplementedError",
        "OSError",
        "OverflowError",
        "PendingDeprecationWarning",
        "ReferenceError",
        "RuntimeError",
        "RuntimeWarning",
        "StandardError",
        "StopIteration",
        "SyntaxError",
        "SyntaxWarning",
        "SystemError",
        "SystemExit",
        "TabError",
        "True",
        "TypeError",
        "UnboundLocalError",
        "UnicodeDecodeError",
        "UnicodeEncodeError",
        "UnicodeError",
        "UnicodeTranslateError",
        "UnicodeWarning",
        "UserWarning",
        "ValueError",
        "Warning",
        "ZeroDivisionError",
        "__debug__",
        "__doc__",
        "__import__",
        "__name__",
        "__package__",
        "abs",
        "all",
        "any",
        "apply",
        "basestring",
        "bin",
        "bool",
        "buffer",
        "bytearray",
        "bytes",
        "callable",
        "chr",
        "classmethod",
        "cmp",
        "coerce",
        "compile",
        "complex",
        "copyright",
        "credits",
        "delattr",
        "dict",
        "dir",
        "divmod",
        "enumerate",
        "eval",
        "execfile",
        "exit",
        "file",
        "filter",
        "float",
        "format",
        "frozenset",
        "getattr",
        "globals",
        "hasattr",
        "hash",
        "help",
        "hex",
        "id",
        "input",
        "int",
        "intern",
        "isinstance",
        "issubclass",
        "iter",
        "len",
        "license",
        "list",
        "locals",
        "long",
        "map",
        "max",
        "memoryview",
        "min",
        "next",
        "object",
        "oct",
        "open",
        "ord",
        "pow",
        "print",
        "property",
        "quit",
        "range",
        "raw_input",
        "reduce",
        "reload",
        "repr",
        "reversed",
        "round",
        "set",
        "setattr",
        "slice",
        "sorted",
        "staticmethod",
        "str",
        "sum",
        "super",
        "tuple",
        "type",
        "unichr",
        "unicode",
        "vars",
        "xrange",
        "zip",
    ]
)
# Source: import builtins; dir(builtins) (Python 3.5.2)
# Note: Removed 'False', 'None', 'True'. They are keyword in Python 3
python3Builtins = set(
    [
        "ArithmeticError",
        "AssertionError",
        "AttributeError",
        "BaseException",
        "BlockingIOError",
        "BrokenPipeError",
        "BufferError",
        "BytesWarning",
        "ChildProcessError",
        "ConnectionAbortedError",
        "ConnectionError",
        "ConnectionRefusedError",
        "ConnectionResetError",
        "DeprecationWarning",
        "EOFError",
        "Ellipsis",
        "EnvironmentError",
        "Exception",
        "FileExistsError",
        "FileNotFoundError",
        "FloatingPointError",
        "FutureWarning",
        "GeneratorExit",
        "IOError",
        "ImportError",
        "ImportWarning",
        "IndentationError",
        "IndexError",
        "InterruptedError",
        "IsADirectoryError",
        "KeyError",
        "KeyboardInterrupt",
        "LookupError",
        "MemoryError",
        "NameError",
        "NotADirectoryError",
        "NotImplemented",
        "NotImplementedError",
        "OSError",
        "OverflowError",
        "PendingDeprecationWarning",
        "PermissionError",
        "ProcessLookupError",
        "RecursionError",
        "ReferenceError",
        "ResourceWarning",
        "RuntimeError",
        "RuntimeWarning",
        "StopAsyncIteration",
        "StopIteration",
        "SyntaxError",
        "SyntaxWarning",
        "SystemError",
        "SystemExit",
        "TabError",
        "TimeoutError",
        "TypeError",
        "UnboundLocalError",
        "UnicodeDecodeError",
        "UnicodeEncodeError",
        "UnicodeError",
        "UnicodeTranslateError",
        "UnicodeWarning",
        "UserWarning",
        "ValueError",
        "Warning",
        "ZeroDivisionError",
        "__build_class__",
        "__debug__",
        "__doc__",
        "__import__",
        "__loader__",
        "__name__",
        "__package__",
        "__spec__",
        "abs",
        "all",
        "any",
        "ascii",
        "bin",
        "bool",
        "bytearray",
        "bytes",
        "callable",
        "chr",
        "classmethod",
        "compile",
        "complex",
        "copyright",
        "credits",
        "delattr",
        "dict",
        "dir",
        "divmod",
        "enumerate",
        "eval",
        "exec",
        "exit",
        "filter",
        "float",
        "format",
        "frozenset",
        "getattr",
        "globals",
        "hasattr",
        "hash",
        "help",
        "hex",
        "id",
        "input",
        "int",
        "isinstance",
        "issubclass",
        "iter",
        "len",
        "license",
        "list",
        "locals",
        "map",
        "max",
        "memoryview",
        "min",
        "next",
        "object",
        "oct",
        "open",
        "ord",
        "pow",
        "print",
        "property",
        "quit",
        "range",
        "repr",
        "reversed",
        "round",
        "set",
        "setattr",
        "slice",
        "sorted",
        "staticmethod",
        "str",
        "sum",
        "super",
        "tuple",
        "type",
        "vars",
        "zip",
    ]
)
# Merge the two sets to get a general Python builtins list
pythonBuiltins = python2Builtins | python3Builtins
# Instance sets
python2Instance = set(["self"])
python3Instance = set(["self"])
pythonInstance = python2Instance | python3Instance
class MultilineStringToken(StringToken):
    """Characters representing a multi-line string."""
    defaultStyle = "fore:#7F0000"
class CellCommentToken(CommentToken):
    """Characters representing a cell separator comment: "##"."""
    defaultStyle = "bold:yes, underline:yes"
# This regexp is used to find special stuff, such as comments, numbers and
# strings.
tokenProg = re.compile(
    "#|"
    + "(["  # Comment or
    + ALPHANUM
    + "_]+)|"
    + "("  # Identifiers/numbers (group 1) or
    + "([bB]|[uU])?"  # Begin of string group (group 2)
    + "[rR]?"  # Possibly bytes or unicode (py2.x)
    + "(\"\"\"|'''|\"|')"  # Possibly a raw string
    + ")|"  # String start (triple qoutes first, group 4)
    + "(\(|\[|\{)|"  # End of string group
    + "(\)|\]|\})|"  # Opening parenthesis (gr 5)
    + "("  # Closing parenthesis (gr 6)
    + chr(160)
    + ")"  # non-breaking space (gr 7)
)
# For a given type of string ( ', " , ''' , """ ),get  the RegExp
# program that matches the end. (^|[^\\]) means: start of the line
# or something that is not \ (since \ is supposed to escape the following
# quote) (\\\\)* means: any number of two slashes \\ since each slash will
# escape the next one
endProgs = {
    "'": re.compile(r"(^|[^\\])(\\\\)*'"),
    '"': re.compile(r'(^|[^\\])(\\\\)*"'),
    "'''": re.compile(r"(^|[^\\])(\\\\)*'''"),
    '"""': re.compile(r'(^|[^\\])(\\\\)*"""'),
}
class PythonParser(Parser):
    """Parser for Python in general."""
    _extensions = []
    _shebangKeywords = []
    # The list of keywords is overridden by the Python2/3 specific parsers
    _keywords = set()
    # The list of builtins and instances is overridden by the Python2/3 specific
parsers
    _builtins = set()
    _instance = set()
    def _identifierState(self, identifier=None):
        """Given an identifier returs the identifier state:
        3 means the current identifier can be a function.
        4 means the current identifier can be a class.
        0 otherwise.
        This method enables storing the state during the line,
        and helps the Cython parser to reuse the Python parser's code.
        """
        if identifier is None:
            # Explicit get/reset
            try:
                state = self._idsState
            except Exception:
                state = 0
            self._idsState = 0
            return state
        elif identifier == "def":
            # Set function state
            self._idsState = 3
            return 3
        elif identifier == "class":
            # Set class state
            self._idsState = 4
            return 4
        else:
            # This one can be func or class, next one can't
            state = self._idsState
            self._idsState = 0
            return state
    def parseLine(self, line, previousState=0):
        """parseLine(line, previousState=0)
        Parse a line of Python code, yielding tokens.
        previousstate is the state of the previous block, and is used
        to handle line continuation and multiline strings.
        """
        line = text_type(line)
        # Init
        pos = 0  # Position following the previous match
        # identifierState and previousstate values:
        # 0: nothing special
        # 1: multiline comment single qoutes
        # 2: multiline comment double quotes
        # 3: a def keyword
        # 4: a class keyword
        # Handle line continuation after def or class
        # identifierState is 3 or 4 if the previous identifier was 3 or 4
        if previousState == 3 or previousState == 4:
            self._identifierState({3: "def", 4: "class"}[previousState])
        else:
            self._identifierState(None)
        if previousState in [1, 2]:
            token = MultilineStringToken(line, 0, 0)
            token._style = ["", "'''", '"""'][previousState]
            tokens = self._findEndOfString(line, token)
            # Process tokens
            for token in tokens:
                yield token
                if isinstance(token, BlockState):
                    return
            pos = token.end
        # Enter the main loop that iterates over the tokens and skips strings
        while True:
            # Get next tokens
            tokens = self._findNextToken(line, pos)
            if not tokens:
                return
            elif isinstance(tokens[-1], StringToken):
                moreTokens = self._findEndOfString(line, tokens[-1])
                tokens = tokens[:-1] + moreTokens
            # Process tokens
            for token in tokens:
                yield token
                if isinstance(token, BlockState):
                    return
            pos = token.end
    def _findEndOfString(self, line, token):
        """_findEndOfString(line, token)
        Find the end of a string. Returns (token, endToken). The first
        is the given token or a replacement (UnterminatedStringToken).
        The latter is None, or the BlockState. If given, the line is
        finished.
        """
        # Set state
        self._identifierState(None)
        # Find the matching end in the rest of the line
        # Do not use the start parameter of search, since ^ does not work then
        style = token._style
        endMatch = endProgs[style].search(line[token.end :])
        if endMatch:
            # The string does end on this line
            tokenArgs = line, token.start, token.end + endMatch.end()
            if style in ['"""', "'''"]:
                token = MultilineStringToken(*tokenArgs)
            else:
                token.end = token.end + endMatch.end()
            return [token]
        else:
            # The string does not end on this line
            tokenArgs = line, token.start, token.end + len(line)
            if style == "'''":
                return [MultilineStringToken(*tokenArgs), BlockState(1)]
            elif style == '"""':
                return [MultilineStringToken(*tokenArgs), BlockState(2)]
            else:
                return [UnterminatedStringToken(*tokenArgs)]
    def _findNextToken(self, line, pos):
        """_findNextToken(line, pos):
        Returns a token or None if no new tokens can be found.
        """
        # Init tokens, if pos too large, were done
        if pos > len(line):
            return None
        tokens = []
        # Find the start of the next string or comment
        match = tokenProg.search(line, pos)
        # Process the Non-Identifier between pos and match.start()
        # or end of line
        nonIdentifierEnd = match.start() if match else len(line)
        # Return the Non-Identifier token if non-null
        # todo: here it goes wrong (allow returning more than one token?)
        token = NonIdentifierToken(line, pos, nonIdentifierEnd)
        strippedNonIdentifier = ustr(token).strip()
        if token:
            tokens.append(token)
        # Do checks for line continuation and identifierState
        # Is the last non-whitespace a line-continuation character?
        if strippedNonIdentifier.endswith("\\"):
            lineContinuation = True
            # If there are non-whitespace characters after def or class,
            # cancel the identifierState
            if strippedNonIdentifier != "\\":
                self._identifierState(None)
        else:
            lineContinuation = False
            # If there are non-whitespace characters after def or class,
            # cancel the identifierState
            if strippedNonIdentifier != "":
                self._identifierState(None)
        # If no match, we are done processing the line
        if not match:
            if lineContinuation:
                tokens.append(BlockState(self._identifierState()))
            return tokens
        # The rest is to establish what identifier we are dealing with
        # Comment
        if match.group() == "#":
            matchStart = match.start()
            if not line[:matchStart].strip() and (
                line[matchStart:].startswith("##")
                or line[matchStart:].startswith("#%%")
                or line[matchStart:].startswith("# %%")
            ):
                tokens.append(CellCommentToken(line, matchStart, len(line)))
            elif self._isTodoItem(line[matchStart + 1 :]):
                tokens.append(TodoCommentToken(line, matchStart, len(line)))
            else:
                tokens.append(CommentToken(line, matchStart, len(line)))
            if lineContinuation:
                tokens.append(BlockState(self._identifierState()))
            return tokens
        # If there are non-whitespace characters after def or class,
        # cancel the identifierState (this time, also if there is just a \
        # since apparently it was not on the end of a line)
        if strippedNonIdentifier != "":
            self._identifierState(None)
        # Identifier ("a word or number") Find out whether it is a key word
        if match.group(1) is not None:
            identifier = match.group(1)
            tokenArgs = line, match.start(), match.end()
            # Set identifier state
            identifierState = self._identifierState(identifier)
            if identifier in self._keywords:
                tokens.append(KeywordToken(*tokenArgs))
            elif identifier in self._builtins and (
                "." + identifier not in line and "def " + identifier not in line
            ):
                tokens.append(BuiltinsToken(*tokenArgs))
            elif identifier in self._instance:
                tokens.append(InstanceToken(*tokenArgs))
            elif identifier[0] in "0123456789":
                self._identifierState(None)
                tokens.append(NumberToken(*tokenArgs))
            else:
                if identifierState == 3 and line[match.end()
:].lstrip().startswith(
                    "("
                ):
                    tokens.append(FunctionNameToken(*tokenArgs))
                elif identifierState == 4:
                    tokens.append(ClassNameToken(*tokenArgs))
                else:
                    tokens.append(IdentifierToken(*tokenArgs))
        elif match.group(2) is not None:
            # We have matched a string-start
            # Find the string style ( ' or " or ''' or """)
            token = StringToken(line, match.start(), match.end())
            token._style = match.group(4)  # The style is in match group 4
            tokens.append(token)
        elif match.group(5) is not None:
            token = OpenParenToken(line, match.start(), match.end())
            token._style = match.group(5)
            tokens.append(token)
        elif match.group(6) is not None:
            token = CloseParenToken(line, match.start(), match.end())
            token._style = match.group(6)
            tokens.append(token)
        elif match.group(7) is not None:
            token = IllegalToken(line, match.start(), match.end())
            token._style = match.group(7)
            tokens.append(token)
        # Done
        return tokens
class PythonParser(PythonParser):  # Ambiguous Python parser
    """Parser for either Python2 or Python3, and we do not know which."""
    _extensions = [".py", ".pyw"]
    _shebangKeywords = ["python"]
    # The list of keywords is overridden by the Python2/3 specific parsers
    _keywords = pythonKeywords
    # The list of builtins and instances is overridden by the Python2/3 specific
parsers
    _builtins = pythonBuiltins
    _instance = pythonInstance
    @classmethod
    def disambiguate(cls, text):
        # try to look into the source...
        # or ... well, ppl should use Python3. Use a shebang to annotate a
Python file as Python 2
        return "python3"
class Python2Parser(PythonParser):
    """Parser for Python 2.x code."""
    # The application should choose whether to set the Py 2 specific parser
    _extensions = []
    _shebangKeywords = ["python2"]
    _keywords = python2Keywords
    _builtins = python2Builtins
    _instance = python2Instance
class Python3Parser(PythonParser):
    """Parser for Python 3.x code."""
    # The application should choose whether to set the Py 3 specific parser
    _extensions = []
    _shebangKeywords = ["python3"]
    _keywords = python3Keywords
    _builtins = python3Builtins
    _instance = python3Instance
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2018, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
from . import Parser, BlockState, text_type
# Import tokens in module namespace
from .tokens import (
    CommentToken,
    StringToken,
    UnterminatedStringToken,
    IdentifierToken,
    ClassNameToken,
    KeywordToken,
    NumberToken,
    OpenParenToken,
    CloseParenToken,
)
class SExprParser(Parser):
    """Parser for S-expressions."""
    _extensions = [".lisp", ".ss", ".sls", ".scm"]
    _keywords = []  # can be overloaded
    def parseLine(self, line, comment_level=0):
        """parseLine(line, comment_level=0)
        Parse a line of code, yielding tokens.
        previousstate is the state of the previous block, and is used
        to handle line continuation and multiline strings.
        """
        line = text_type(line)
        if comment_level < 0:
            comment_level = 0
        if comment_level > 0:
            token = CommentToken(line, 0, 0)
        pos = 0
        while pos < len(line):
            pos = self._skip_whitespace(line, pos)
            if pos >= len(line):
                break
            # Parse block comments
            if line[pos] == "(" and pos < len(line) - 1 and line[pos + 1] ==
";":
                if comment_level == 0:
                    token = CommentToken(line, pos, pos)
                comment_level += 1
                pos += 1
            elif line[pos] == ";" and pos < len(line) - 1 and line[pos + 1] ==
")":
                if comment_level == 1:
                    token.end = pos + 2
                    yield token
                comment_level = max(0, comment_level - 1)
                pos += 1
            elif comment_level > 0:
                pos += 1
            else:
                # Outside of block comments ...
                if line[pos] == ";" and pos < len(line) - 1 and line[pos + 1] ==
";":
                    yield CommentToken(line, pos, len(line))
                    pos = len(line)
                elif line[pos] == "(":
                    token = OpenParenToken(line, pos, pos + 1)
                    token._style = "("
                    yield token
                    pos += 1
                elif line[pos] == ")":
                    token = CloseParenToken(line, pos, pos + 1)
                    token._style = ")"
                    yield token
                    pos += 1
                elif line[pos] == '"':
                    i0 = pos
                    esc = False
                    for i in range(i0 + 1, len(line)):
                        if not esc and line[i] == '"':
                            pos = i + 1
                            yield StringToken(line, i0, pos)
                            break
                        esc = line[i] == "\\"
                    else:
                        yield UnterminatedStringToken(line, i0, len(line))
                        pos = len(line)
                else:
                    # word: number, keyword or normal identifier
                    i0 = pos
                    for i in range(i0, len(line)):
                        if line[i] in " \t\r\n)":
                            yield self._get_token_for_word(line, i0, i)
                            pos = i
                            break
                    else:
                        pos = len(line)
                        yield self._get_token_for_word(line, i0, len(line))
        if comment_level > 0:
            token.end = len(line)
            yield token
        yield BlockState(comment_level)
    def _skip_whitespace(self, line, pos):
        while pos < len(line):
            if line[pos] not in " \t\r\n":
                break
            pos += 1
        return pos
    def _get_token_for_word(self, line, i0, i1):
        word = line[i0:i1]
        is_number = False
        try:
            float(word)
            is_number = True
        except ValueError:
            pass
        if is_number or word.startswith("$"):
            return NumberToken(line, i0, i1)
        elif word in self._keywords:  # highlight extra
            return ClassNameToken(line, i0, i1)  # ClassNameToken or 
FunctionNameToken
        elif i0 > 0 and line[i0 - 1] == "(":  # First element in expression is 
"keyword"
            return KeywordToken(line, i0, i1)
        else:
            return IdentifierToken(line, i0, i1)
class WatParser(SExprParser):
    """Parser for textual WASM (WAT) code."""
    _extensions = [".wat", ".wast"]
    _keywords = [
        "module",
        "type",
        "import",
        "func",
        "table",
        "memory",
        "global",
        "export",
        "start",
        "element",
        "data",
    ]
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module tokens
Defines the base Token class and a few generic tokens.
Tokens are used by parsers to identify for groups of characters
what they represent. This is in turn used by the highlighter
to determine how these characters should be styled.
"""
# Many parsers need this
ALPHANUM = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"
from ..style import StyleFormat, StyleElementDescription
from ..misc import ustr
class Token(object):
    """Token(line, start, end)
    Base token class.
    A token is a group of characters representing "something".
    What is represented, is specified by the subclass.
    Each token class should have a docstring describing the meaning
    of the characters it is applied to.
    """
    defaultStyle = "fore:#000, bold:no, underline:no, italic:no"
    isToken = True  # For the BlockState object, which is also returned by the
parsers, this is False
    def __init__(self, line="", start=0, end=0):
        self.line = ustr(line)
        self.start = start
        self.end = end
        self._name = self._getName()
    def __str__(self):  # on 2.x we use __unicode__
        return self.line[self.start : self.end]
    def __unicode__(self):  # for py 2.x
        return self.line[self.start : self.end]
    def __repr__(self):
        return repr("%s:%s" % (self.name, self))
    def __len__(self):
        # Defining a length also gives a Token a boolean value: True if there
        # are any characters (len!=0) and False if there are none
        return self.end - self.start
    def _getName(self):
        """Get the name of this token."""
        nameParts = ["Syntax"]
        if "_parser" in self.__module__:
            language = self.__module__.split("_")[0]
            language = language.split(".")[-1]
            nameParts.append(language[0].upper() + language[1:])
        nameParts.append(self.__class__.__name__[:-5].lower())
        return ".".join(nameParts)
    def getDefaultStyleFormat(self):
        elements = []
        def collect(cls):
            if hasattr(cls, "defaultStyle"):
                elements.append(cls.defaultStyle)
                for c in cls.__bases__:
                    collect(c)
        collect(self.__class__)
        se = StyleFormat()
        for e in reversed(elements):
            se.update(e)
        return se
    @property
    def name(self):
        """The name of this token. Used to identify it and attach a style."""
        return self._name
    @property
    def description(self):
        """description()
        Returns a StyleElementDescription instance that describes the
        style element that this token represents.
        """
        format = self.getDefaultStyleFormat()
        des = "syntax: " + self.__doc__
        return StyleElementDescription(self.name, des, str(format))
class CommentToken(Token):
    """Characters representing a comment in the code."""
    defaultStyle = "fore:#007F00"
class TodoCommentToken(CommentToken):
    """Characters representing a comment in the code."""
    defaultStyle = "fore:#E00,italic"
class StringToken(Token):
    """Characters representing a textual string in the code."""
    defaultStyle = "fore:#7F007F"
class UnterminatedStringToken(StringToken):
    """Characters belonging to an unterminated string."""
    defaultStyle = "underline:dotted"
# todo: request from user: whitespace token
class TextToken(Token):
    """Anything that is not a string or comment."""
    defaultStyle = "fore:#000"
class IdentifierToken(TextToken):
    """Characters representing normal text (i.e. words)."""
    defaultStyle = ""
class NonIdentifierToken(TextToken):
    """Not a word (operators, whitespace, etc.)."""
    defaultStyle = ""
class KeywordToken(IdentifierToken):
    """A keyword is a word with a special meaning to the language."""
    defaultStyle = "fore:#00007F, bold:yes"
class BuiltinsToken(IdentifierToken):
    """Characters representing a builtins in the code."""
    defaultStyle = ""
class InstanceToken(IdentifierToken):
    """Characters representing a instance in the code."""
    defaultStyle = ""
class NumberToken(IdentifierToken):
    """Characters represening a number."""
    defaultStyle = "fore:#007F7F"
class FunctionNameToken(IdentifierToken):
    """Characters represening the name of a function."""
    defaultStyle = "fore:#007F7F, bold:yes"
class ClassNameToken(IdentifierToken):
    """Characters represening the name of a class."""
    defaultStyle = "fore:#0000FF, bold:yes"
class ParenthesisToken(TextToken):
    """Parenthesis (and square and curly brackets)."""
    defaultStyle = ""
class OpenParenToken(ParenthesisToken):
    """Opening parenthesis (and square and curly brackets)."""
    defaultStyle = ""
class CloseParenToken(ParenthesisToken):
    """Closing parenthesis (and square and curly brackets)."""
    defaultStyle = ""
class IllegalToken(Token):
    """Illegal tokens, eg. NBSP  ."""
    defaultStyle = "back:#ffd7c7"
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the codeeditor development team
#
# Codeeditor is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Subpackage parsers
This subpackage contains all the syntax parsers for the
different languages.
"""
""" CREATING PARSERS
Making a parser requires these things:
  * Place a module in the parsers directory, which has a name
    ending in "_parser.py"
  * In the module implement one or more classes that inherit
    from ..parsers.Parser (or a derived class), and
    implement the parseLine method.
  * The module should import all the tokens in whiches to use
    from ..parsers.tokens. New tokens can also be
    defined by subclassing one of the token classes.
  * In codeeditor/parsers/__init__.py, add the new module to the
    list of imported parsers.
"""
import sys
from . import tokens
if sys.version_info[0] >= 3:
    text_type = str
else:
    text_type = unicode  # noqa
class BlockState(object):
    """BlockState(state=0, info=None)
    The blockstate object should be used by parsers to
    return the block state of the processed line.
    This would typically be the last item to be yielded, but this
    it may also be yielded befor the last yielded token. One can even
    yield multiple of these items, in which case the last one considered
    valid.
    """
    isToken = False
    def __init__(self, state=0, info=None):
        self._state = int(state)
        self._info = info
    @property
    def state(self):
        """The integer value representing the block state."""
        return self._state
    @property
    def info(self):
        """Get the information corresponding to the block."""
        return self._info
# Base parser class (needs to be defined before importing parser modules)
class Parser(object):
    """Base parser class.
    All parsers should inherit from this class.
    This base class generates a 'TextToken' for each line
    """
    _extensions = []
    _shebangKeywords = []
    _keywords = []
    @classmethod
    def getParserName(cls):
        name = cls.__name__
        if name.endswith("Parser") and len(name) >= 6:
            name = name[:-6].lower()
        return name
    @classmethod
    def disambiguate(cls, text):
        return cls.getParserName()
    def parseLine(self, line, previousState=0):
        """parseLine(line, previousState=0)
        The method that should be implemented by the parser. The
        previousState argument can be used to determine how
        the previous block ended (e.g. for multiline comments). It
        is an integer, the meaning of which is only known to the
        specific parser.
        This method should yield token instances. The last token can
        be a BlockState to specify the previousState for the
        next block.
        """
        yield tokens.TextToken(line, 0, len(line))
    def name(self):
        """name()
        Get the name of the parser.
        """
        name = self.__class__.__name__.lower()
        if name.endswith("parser"):
            name = name[:-6]
        return name
    def __repr__(self):
        """String representation of the parser."""
        return '<Parser for "%s">' % self.name()
    def keywords(self):
        """keywords()
        Get a list of keywords valid for this parser.
        """
        return [k for k in self._keywords]
    def filenameExtensions(self):
        """filenameExtensions()
        Get a list of filename extensions for which this parser
        is appropriate.
        """
        return ["." + e.lstrip(".").lower() for e in self._extensions]
    def shebangKeywords(self):
        """shebangKeywords()
        Get a list of shebang keywords for which this parser
        is appropriate.
        """
        return self._shebangKeywords.copy()
    def getStyleElementDescriptions(cls):
        """getStyleElementDescriptions()
        This method returns a list of the StyleElementDescription
        instances used by this parser.
        """
        descriptions = {}
        for token in cls.getUsedTokens(cls):
            descriptions[token.description.key] = token.description
        return list(descriptions.values())
    def getUsedTokens(self):
        """getUsedTokens()
        Get a a list of token instances used by this parser.
        """
        # Get module object of the parser
        try:
            mod = sys.modules[self.__module__]
        except KeyError:
            return []
        # Get token classes from module
        tokenClasses = []
        for name in mod.__dict__:
            member = mod.__dict__[name]
            if isinstance(member, type) and issubclass(member, tokens.Token):
                if member is not tokens.Token:
                    tokenClasses.append(member)
        # Return as instances
        return [t() for t in tokenClasses]
    def _isTodoItem(self, text):
        """_isTodoItem(text)
        Get whether the given text (which should be a comment) represents
        a todo item. Todo items start with "todo", "2do" or "fixme",
        optionally with a colon at the end.
        """
        # Get first word
        word = text.lstrip().split(" ", 1)[0].rstrip(":")
        # Test
        if word.lower() in ["todo", "2do", "fixme"]:
            return True
        else:
            return False
## Import parsers statically
# We could load the parser dynamically from the source files in the
# directory, but this takes quite some effort to get righ when apps
# are frozen. This is doable (I do it in Visvis) but it requires the
# user to specify the parser modules by hand when freezing an app.
#
# In summary: it takes a lot of trouble, which can be avoided by just
# listing all parsers here.
from . import python_parser  # noqa
from . import cython_parser  # noqa
from . import c_parser  # noqa
from . import s_expr_parser  # noqa
```

```
import os
import sys
from pyzo.qt import QtCore, QtGui, QtWidgets
from pyzo import qt
import pyzo
from pyzo.util import paths
class AboutDialog(QtWidgets.QDialog):
    def __init__(self, parent):
        QtWidgets.QDialog.__init__(self, parent)
        self.setWindowTitle(pyzo.translate("menu dialog", "About Pyzo"))
        self.resize(600, 500)
        # Layout
        layout = QtWidgets.QVBoxLayout(self)
        self.setLayout(layout)
        # Create image and title
        im = QtGui.QPixmap(
            os.path.join(pyzo.pyzoDir, "resources", "appicons",
"pyzologo64.png")
        )
        imlabel = QtWidgets.QLabel(self)
        imlabel.setPixmap(im)
        textlabel = QtWidgets.QLabel(self)
        textlabel.setText("<h3>Pyzo: the Interactive Editor for Python</h3>")
        #
        titleLayout = QtWidgets.QHBoxLayout()
        titleLayout.addWidget(imlabel, 0)
        titleLayout.addWidget(textlabel, 1)
        #
        layout.addLayout(titleLayout, 0)
        # Create tab bar
        self._tabs = QtWidgets.QTabWidget(self)
        self._tabs.setDocumentMode(True)
        layout.addWidget(self._tabs, 1)
        # Create button box
        self._butBox = QtWidgets.QDialogButtonBox(self)
        self._butBox.setOrientation(QtCore.Qt.Horizontal)
        self._butBox.setStandardButtons(self._butBox.Close)
        layout.addWidget(self._butBox, 0)
        # Signals
        self._butBox.rejected.connect(self.close)
        # Create tabs
        self.createGeneralTab()
    def addTab(self, title, text, rich=True):
        # Create label to show info
        label = QtWidgets.QTextEdit(self)
        label.setLineWrapMode(label.WidgetWidth)
        label.setReadOnly(True)
        # Set text
        if rich:
            label.setHtml(text)
        else:
            label.setText(text)
        # Add to tab bar
        self._tabs.addTab(label, title)
        # Done
        return label
    def createGeneralTab(self):
        aboutText = """
        {}<br><br>
        <b>Version info</b><br>
        Pyzo version: <u>{}</u><br>
        Platform: {}<br>
        Python version: {}<br>
        Qt version: {}<br>
        {} version: {}<br>
        <br>
        <b>Pyzo directories</b><br>
        Pyzo source directory: {}<br>
        Pyzo config directory: {}<br>
        Pyzo userdata directory: {}<br>
        <br>
        <b>License</b><br>
        Pyzo is open source, it's code is distributed under the 2-Clause BSD
license.
        <br><br>
        <b>Contributors</b><br>
        Pyzo is coded with ♥ by Almar Klein and over 30 contributors:<br>
        https://github.com/pyzo/pyzo/graphs/contributors
        <br><br>
        <b>Acknowledgements</b><br>
        Pyzo is written in Python 3 and uses the Qt widget
        toolkit. Pyzo uses code and concepts that are inspired by
        IPython, Pype, and Spyder.
        Pyzo uses a (modified) subset of the silk icon set,
        by Mark James (http://www.famfamfam.com/lab/icons/silk/).
        """
        # Determine if this is PyQt or Pyside
        qtWrapper = qt.API_NAME
        qtVersion = qt.QT_VERSION
        qtWrapperVersion = qt.PYSIDE_VERSION or qt.PYQT_VERSION
        # Insert information texts
        if paths.is_frozen():
            versionText = pyzo.__version__ + " (binary)"
        else:
            versionText = pyzo.__version__ + " (source)"
        aboutText = aboutText.format(
            "Pyzo - Python to the people!",
            versionText,
            sys.platform,
            sys.version.split(" ")[0],
            qtVersion,
            qtWrapper,
            qtWrapperVersion,
            pyzo.pyzoDir,
            pyzo.appConfigDir,
            pyzo.appDataDir,
        )
        self.addTab("General", aboutText)
if __name__ == "__main__":
    # pyzo.license = {'name': 'AK', 'company': ''}
    m = AboutDialog(None)
    m.show()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
# Author: Windel Bouwman
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
"""
Tool that can view qt help files via the qthelp engine.
Run make_docs.sh from:
https://bitbucket.org/windel/qthelpdocs
Copy the "docs" directory to the pyzo root!
"""
from pyzo.qt import QtCore, QtGui, QtWidgets  # noqa
from pyzo import getResourceDirs
import os
help_help = """
<h1>Documentation</h1>
<p>
Welcome to the Pyzo assistant. Pyzo uses the Qt Help system for documentation.
This is also used by the Qt Assistant. You can use this viewer
to view documentation provided by other projects.
</p>
<h2>Add documentation</h2>
<p>
To add documentation to Pyzo, go to the settings tab and select add. Then
select a Qt Compressed Help file (*.qch). qch-files can be found in the Qt
installation directory (for example in /usr/share/doc/qt under linux). For
other projects you can download pre-build qch files from here:
<a href="https://github.com/windelbouwman/qthelpdocs/releases">https://github.com/windelbouwman/qthelpdocs/releases</a>.
</p>
<p>
<strong>Note</strong>
When a documentation file is added, it is not copied into the pyzo settings
dir, so you have to leave this file in place.
</p>
"""
tool_name = "Assistant"
tool_summary = "Browse qt help documents"
class Settings(QtWidgets.QWidget):
    def __init__(self, engine):
        super().__init__()
        self._engine = engine
        layout = QtWidgets.QVBoxLayout(self)
        add_button = QtWidgets.QPushButton("Add")
        del_button = QtWidgets.QPushButton("Delete")
        self._view = QtWidgets.QListView()
        layout.addWidget(self._view)
        layout2 = QtWidgets.QHBoxLayout()
        layout2.addWidget(add_button)
        layout2.addWidget(del_button)
        layout.addLayout(layout2)
        self._model = QtCore.QStringListModel()
        self._view.setModel(self._model)
        self._model.setStringList(self._engine.registeredDocumentations())
        add_button.clicked.connect(self.add_doc)
        del_button.clicked.connect(self.del_doc)
    def add_doc(self):
        doc_file = QtWidgets.QFileDialog.getOpenFileName(
            self,
            "Select a compressed help file",
            filter="Qt compressed help files (*.qch)",
        )
        if isinstance(doc_file, tuple):
            doc_file = doc_file[0]
        self.add_doc_do(doc_file)
    def add_doc_do(self, doc_file):
        ok = self._engine.registerDocumentation(doc_file)
        if ok:
            self._model.setStringList(self._engine.registeredDocumentations())
        else:
            QtWidgets.QMessageBox.critical(self, "Error", "Error loading doc")
    def del_doc(self):
        idx = self._view.currentIndex()
        if idx.isValid():
            doc_file = self._model.data(idx, QtCore.Qt.DisplayRole)
            self.del_doc_do(doc_file)
    def del_doc_do(self, doc_file):
        self._engine.unregisterDocumentation(doc_file)
        self._model.setStringList(self._engine.registeredDocumentations())
class HelpBrowser(QtWidgets.QTextBrowser):
    """Override textbrowser to implement load resource"""
    def __init__(self, engine):
        super().__init__()
        self._engine = engine
        # Override default navigation behavior:
```

```
        self.anchorClicked.connect(self.handle_url)
        self.setOpenLinks(False)
    def handle_url(self, url):
        """Open external urls not in this viewer"""
        if url.scheme() in ["http", "https"]:
            QtGui.QDesktopServices.openUrl(url)
        else:
            self.setSource(url)
    def loadResource(self, typ, url):
        if url.scheme() == "qthelp":
            return self._engine.fileData(url)
        else:
            return super().loadResource(typ, url)
class PyzoAssistant(QtWidgets.QWidget):
    """
    Show help contents and browse qt help files.
    """
    def __init__(self, parent=None, collection_filename=None):
        """
        Initializes an assistance instance.
        When collection_file is none, it is determined from the
        appDataDir.
        """
        from pyzo.qt import QtHelp
        super().__init__(parent)
        self.setWindowTitle("Help")
        pyzoDir, appDataDir, appConfigDir = getResourceDirs()
        if collection_filename is None:
            # Collection file is stored in pyzo data dir:
            collection_filename = os.path.join(appDataDir, "tools", "docs.qhc")
        self._engine = QtHelp.QHelpEngine(collection_filename)
        # Important, call setup data to load the files:
        self._engine.setupData()
        # If no files are loaded, register at least the pyzo docs:
        if len(self._engine.registeredDocumentations()) == 0:
            doc_file = os.path.join(pyzoDir, "resources", "pyzo.qch")
            self._engine.registerDocumentation(doc_file)
        # The main players:
        self._content = self._engine.contentWidget()
        self._index = self._engine.indexWidget()
        self._indexTab = QtWidgets.QWidget()
        il = QtWidgets.QVBoxLayout(self._indexTab)
        filter_text = QtWidgets.QLineEdit()
        il.addWidget(filter_text)
        il.addWidget(self._index)
        self._helpBrowser = HelpBrowser(self._engine)
        self._searchEngine = self._engine.searchEngine()
        self._settings = Settings(self._engine)
        self._progress = QtWidgets.QWidget()
        pl = QtWidgets.QHBoxLayout(self._progress)
        bar = QtWidgets.QProgressBar()
        bar.setMaximum(0)
        pl.addWidget(QtWidgets.QLabel("Indexing"))
        pl.addWidget(bar)
        self._searchResultWidget = self._searchEngine.resultWidget()
        self._searchQueryWidget = self._searchEngine.queryWidget()
        self._searchTab = QtWidgets.QWidget()
        search_layout = QtWidgets.QVBoxLayout(self._searchTab)
        search_layout.addWidget(self._searchQueryWidget)
        search_layout.addWidget(self._searchResultWidget)
        tab = QtWidgets.QTabWidget()
        tab.addTab(self._content, "Contents")
        tab.addTab(self._indexTab, "Index")
        tab.addTab(self._searchTab, "Search")
        tab.addTab(self._settings, "Settings")
        splitter = QtWidgets.QSplitter(self)
        splitter.addWidget(tab)
        splitter.addWidget(self._helpBrowser)
        layout = QtWidgets.QVBoxLayout(self)
        layout.addWidget(splitter)
        layout.addWidget(self._progress)
        # Connect clicks:
        self._content.linkActivated.connect(self._helpBrowser.setSource)
        self._index.linkActivated.connect(self._helpBrowser.setSource)
        self._searchEngine.searchingFinished.connect(self.onSearchFinish)
        self._searchEngine.indexingStarted.connect(self.onIndexingStarted)
        self._searchEngine.indexingFinished.connect(self.onIndexingFinished)
        filter_text.textChanged.connect(self._index.filterIndices)
self._searchResultWidget.requestShowLink.connect(self._helpBrowser.setSource)
        self._searchQueryWidget.search.connect(self.goSearch)
        # Always re-index on startup:
        self._searchEngine.reindexDocumentation()
        self._search_term = None
        # Show initial page:
        # self.showHelpForTerm('welcome to pyzo')
        self._helpBrowser.setHtml(help_help)
```

```
    def goSearch(self):
        query = self._searchQueryWidget.query()
        self._searchEngine.search(query)
    def onIndexingStarted(self):
        self._progress.show()
    def onIndexingFinished(self):
        self._progress.hide()
    def find_best_page(self, hits):
        if self._search_term is None:
            url, _ = hits[0]
            return url
        try:
            # Try to find max with fuzzy wuzzy:
            from fuzzywuzzy import fuzz
            url, title = max(
                hits, key=lambda hit: fuzz.ratio(hit[1], self._search_term)
            )
            return url
        except ImportError:
            pass
        # Find exact page title:
        for url2, page_title in hits:
            if page_title == self._search_term:
                url = url2
                return url
        for url2, page_title in hits:
            if self._search_term in page_title:
                url = url2
                return url
        # Pick first hit:
        url, _ = hits[0]
        return url
    def onSearchFinish(self, hits):
        if hits == 0:
            return
        hits = self._searchEngine.hits(0, hits)
        if not hits:
            return
        url = self.find_best_page(hits)
        self._helpBrowser.setSource(QtCore.QUrl(url))
    def showHelpForTerm(self, name):
        from pyzo.qt import QtHelp
        # Cache for later use:
```

```
        self._search_term = name
        # Create a query:
        query = QtHelp.QHelpSearchQuery(QtHelp.QHelpSearchQuery.DEFAULT, [name])
        self._searchEngine.search([query])
if __name__ == "__main__":
    app = QtWidgets.QApplication([])
    view = PyzoAssistant()
    view.show()
    app.exec()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module baseTextCtrl
Defines the base text control to be inherited by the shell and editor
classes. Implements styling, introspection and a bit of other stuff that
is common for both shells and editors.
"""
import pyzo
import os, time
from pyzo.core.pyzoLogging import print
import pyzo.codeeditor.parsers.tokens as Tokens
from pyzo.qt import QtCore, QtGui, QtWidgets
qt = QtGui
# Define style stuff
subStyleStuff = {}
# subStyleStuff = {   'face': Qsci.QsciScintillaBase.SCI_STYLESETFONT ,
#                    'fore': Qsci.QsciScintillaBase.SCI_STYLESETFORE,
#                    'back': Qsci.QsciScintillaBase.SCI_STYLESETBACK,
#                    'size': Qsci.QsciScintillaBase.SCI_STYLESETSIZE,
#                    'bold': Qsci.QsciScintillaBase.SCI_STYLESETBOLD,
#                    'italic': Qsci.QsciScintillaBase.SCI_STYLESETITALIC,
#                    'underline': Qsci.QsciScintillaBase.SCI_STYLESETUNDERLINE}
def normalizePath(path):
    """Normalize the path given.
    All slashes will be made the same (and doubles removed)
    The real case as stored on the file system is recovered.
    Returns None on error.
    """
    # normalize
    path = os.path.abspath(path)  # make sure it is defined from the drive up
    path = os.path.normpath(path)
    # If does not exist, return as is.
    # This also happens if the path's case is incorrect and the
    # file system is case sensitive. That's ok, because the stuff we
    # do below is intended to get the path right on case insensitive
    # file systems.
    if not os.path.isfile(path):
        return path
    # split drive name from the rest
    drive, rest = os.path.splitdrive(path)
    fullpath = drive.upper() + os.sep
    # make lowercase and split in parts
    parts = rest.lower().split(os.sep)
    parts = [part for part in parts if part]
    for part in parts:
        options = [x for x in os.listdir(fullpath) if x.lower() == part]
        if len(options) > 1:
            print("Error normalizing path: Ambiguous path names!")
            return path
        elif not options:
            print("Invalid path (part %s) in %s" % (part, fullpath))
            return path
        fullpath = os.path.join(fullpath, options[0])
    # remove last sep
    return fullpath
def parseLine_autocomplete(tokens):
    """Given a list of tokens (from start to cursor position)
    returns a tuple (base, name).
    autocomp_parse("eat = banan") -> "", "banan"
      ...("eat = food.fruit.ban") -> "food.fruit", "ban"
    When no match found, both elements are an empty string.
    """
    if not len(tokens):
        return "", ""
    if isinstance(tokens[-1], Tokens.NonIdentifierToken) and str(tokens[-1]) ==
".":
        name = ""
    elif isinstance(tokens[-1], (Tokens.IdentifierToken, Tokens.KeywordToken)):
        name = str(tokens[-1])
    else:
        return "", ""
    needle = ""
    # Now go through the remaining tokens in reverse order
    for token in tokens[-2::-1]:
        if isinstance(token, Tokens.NonIdentifierToken) and str(token) == ".":
            needle = str(token) + needle
        elif isinstance(token, (Tokens.IdentifierToken, Tokens.StringToken)):
            needle = str(token) + needle
        else:
            break
    if needle.endswith("."):
        needle = needle[:-1]
    return needle, name
```

```
def parseLine_signature(tokens):
    """Given a list of tokens (from start to cursor position)
    returns a tuple (name, needle, stats).
    stats is another tuple:
    - location of end bracket
    - amount of kommas till cursor (taking nested brackets into account)
    """
    openBraces = []  # Positions at which braces are opened
    for token in tokens:
        if not isinstance(
            token,
            (Tokens.NonIdentifierToken, Tokens.OpenParenToken,
Tokens.CloseParenToken),
        ):
            continue
        for i, c in enumerate(str(token)):
            if c == "(":
                openBraces.append(token.start + i)
            elif c == ")":
                if len(openBraces):
                    openBraces.pop()
    if len(openBraces):
        i = openBraces[-1]
        # Now trim the token list up to (but not inculding) position of
openBraces
        tokens = list(filter(lambda token: token.start < i, tokens))
        # Trim the last token
        if len(tokens):
            tokens[-1].end = i
        name, needle = parseLine_autocomplete(tokens)
        return name, needle, (i, 0)  # TODO: implement stats
    else:
        return "", "", (0, 0)
def makeBytes(text):
    """Make sure the argument is bytes, converting with UTF-8 encoding
    if it is a string."""
    if isinstance(text, bytes):
        return text
    elif isinstance(text, str):
        return text.encode("utf-8")
    else:
        raise ValueError("Expected str or bytes!")
_allScintillas = []
```

```
def getAllScintillas():
    """Get a list of all the scintialla editing components that
    derive from BaseTextCtrl. Used mainly by the menu.
    """
    for i in reversed(range(len(_allScintillas))):
        e = _allScintillas[i]()
        if e is None:
            _allScintillas.pop(i)
        else:
            yield e
pyzo.getAllScintillas = getAllScintillas
from pyzo import codeeditor
class BaseTextCtrl(codeeditor.CodeEditor):
    """The base text control class.
    Inherited by the shell class and the Pyzo editor.
    The class implements autocompletion, calltips, and auto-help
    Inherits from QsciScintilla. I tried to clean up the rather dirty api
    by using more sensible names. Hereby I apply the following rules:
    - if you set something, the method starts with "set"
    - if you get something, the method starts with "get"
    - a position is the integer position fron the start of the document
    - a linenr is the number of a line, an index the position on that line
    - all the above indices apply to the bytes (encoded utf-8) in which the
      text is stored. If you have unicode text, they do not apply!
    - the method name mentions explicityly what you get. getBytes() returns the
      bytes of the document, getString() gets the unicode string that it
      represents. This applies to the get-methods. the set-methods use the
      term text, and automatically convert to bytes using UTF-8 encoding
      when a string is given.
    """
    def __init__(self, *args, **kwds):
        super().__init__(*args, **kwds)
        # Set style/theme
        try:
            theme = pyzo.themes[pyzo.config.settings.theme.lower()]["data"]
            self.setStyle(theme)
            # autocomplete popup theme
            if pyzo.config.view.get("autoComplete_withTheme", False):
                editor_text_theme = theme["editor.text"].split(",")
                popup_background = editor_text_theme[1].split(":")[1]
                popup_text = editor_text_theme[0].split(":")[1]
                autoComplete_theme = "color: {}; background-color:{};".format(
                    popup_text, popup_background
                )
                self.completer().popup().setStyleSheet(autoComplete_theme)
        except Exception as err:
            print("Could not load theme: " + str(err))
        # Set font and zooming
        self.setFont(pyzo.config.view.fontname)
        self.setZoom(pyzo.config.view.zoom)
        self.setShowWhitespace(pyzo.config.view.showWhitespace)
self.setHighlightMatchingBracket(pyzo.config.view.highlightMatchingBracket)
        # Create timer for autocompletion delay
        self._delayTimer = QtCore.QTimer(self)
        self._delayTimer.setSingleShot(True)
        self._delayTimer.timeout.connect(self._introspectNow)
        # For buffering autocompletion and calltip info
        self._callTipBuffer_name = ""
        self._callTipBuffer_time = 0
        self._callTipBuffer_result = ""
        self._autoCompBuffer_name = ""
        self._autoCompBuffer_time = 0
        self._autoCompBuffer_result = []
        self.setAutoCompletionAcceptKeysFromStr(
            pyzo.config.settings.autoComplete_acceptKeys
        )
        self.completer().highlighted.connect(self.updateHelp)
        self.setIndentUsingSpaces(pyzo.config.settings.defaultIndentUsingSpaces)
        self.setIndentWidth(pyzo.config.settings.defaultIndentWidth)
        self.setAutocompletPopupSize(*pyzo.config.view.autoComplete_popupSize)
        self.setAutocompleteMinChars(pyzo.config.settings.autoComplete_minChars)
        self.setCancelCallback(self.restoreHelp)
    def setAutoCompletionAcceptKeysFromStr(self, keys):
        """Set the keys that can accept an autocompletion from a comma delimited
string."""
        # Set autocomp accept key to default if necessary.
        # We force it to be string (see issue 134)
        if not isinstance(keys, str):
            keys = "Tab"
        # Split
        keys = keys.replace(",", " ").split(" ")
        keys = [key for key in keys if key]
        # Set autocomp accept keys
        qtKeys = []
        for key in keys:
            if len(key) > 1:
                key = "Key_" + key[0].upper() + key[1:].lower()
                qtkey = getattr(QtCore.Qt, key, None)
            else:
                qtkey = ord(key)
            if qtkey:
                qtKeys.append(qtkey)
        if QtCore.Qt.Key_Enter in qtKeys and QtCore.Qt.Key_Return not in qtKeys:
            qtKeys.append(QtCore.Qt.Key_Return)
        self.setAutoCompletionAcceptKeys(*qtKeys)
    def _isValidPython(self):
        """_isValidPython()
        Check if the code at the cursor is valid python:
        - the active lexer is the python lexer
        - the style at the cursor is "default"
        """
        # TODO:
        return True
    def getTokensUpToCursor(self, cursor):
        # In order to find the tokens, we need the userState from the
highlighter
        if cursor.block().previous().isValid():
            previousState = cursor.block().previous().userState()
        else:
            previousState = 0
        text = cursor.block().text()[: cursor.positionInBlock()]
        return (
            text,
            list(
                filter(
                    lambda token: token.isToken,  # filter to remove BlockStates
                    self.parser().parseLine(text, previousState),
                )
            ),
        )
    def introspect(self, tryAutoComp=False, delay=True):
        """introspect(tryAutoComp=False, delay=True)
        The starting point for introspection (autocompletion and calltip).
        It will always try to produce a calltip. If tryAutoComp is True,
        will also try to produce an autocompletion list (which, on success,
        will hide the calltip).
        This method will obtain the line and (re)start a timer that will
        call _introspectNow() after a short while. This way, if the
        user types a lot of characters, there is not a stream of useless
        introspection attempts; the introspection is only really started
        after he stops typing for, say 0.1 or 0.5 seconds (depending on
        pyzo.config.autoCompDelay).
        The method _introspectNow() will parse the line to obtain
        information required to obtain the autocompletion and signature
        information. Then it calls processCallTip and processAutoComp
        which are implemented in the editor and shell classes.
        """
        # Find the tokens up to the cursor
        cursor = self.textCursor()
        text, tokensUptoCursor = self.getTokensUpToCursor(cursor)
        # TODO: Only proceed if valid python (no need to check for comments/
        # strings, this is done by the processing of the tokens). Check for
python style
        # Is the char valid for auto completion?
        if tryAutoComp:
            if not text or not (text[-1] in (Tokens.ALPHANUM + "._")):
                self.autocompleteCancel()
                tryAutoComp = False
        # Store line and (re)start timer
        cursor.setKeepPositionOnInsert(True)
        self._delayTimer._tokensUptoCursor = tokensUptoCursor
        self._delayTimer._cursor = cursor
        self._delayTimer._tryAutoComp = tryAutoComp
        if delay:
            self._delayTimer.start(pyzo.config.advanced.autoCompDelay)
        else:
            self._delayTimer.start(1)  # self._introspectNow()
    def _introspectNow(self):
        """This method is called a short while after introspect()
        by the timer. It parses the line and calls the specific methods
        to process the callTip and autoComp.
        """
        tokens = self._delayTimer._tokensUptoCursor
        if pyzo.config.settings.autoCallTip:
            # Parse the line, to get the name of the function we should calltip
            # if the name is empty/None, we should not show a signature
            name, needle, stats = parseLine_signature(tokens)
            if needle:
                # Compose actual name
                fullName = needle
                if name:
                    fullName = name + "." + needle
                # Process
                offset = (
                    self._delayTimer._cursor.positionInBlock() - stats[0] +
len(needle)
                )
                cto = CallTipObject(self, fullName, offset)
                self.processCallTip(cto)
            else:
                self.calltipCancel()
        if self._delayTimer._tryAutoComp and pyzo.config.settings.autoComplete:
            # Parse the line, to see what (partial) name we need to complete
            name, needle = parseLine_autocomplete(tokens)
            if name or needle:
                # Try to do auto completion
                aco = AutoCompObject(self, name, needle)
                self.processAutoComp(aco)
    def processCallTip(self, cto):
        """Overridden in derive class"""
        pass
    def processAutoComp(self, aco):
        """Overridden in derive class"""
        pass
    def _onDoubleClick(self):
        """When double clicking on a name, autocomplete it."""
        self.processHelp(addToHist=True)
    def helpOnText(self, pos):
        hw = pyzo.toolManager.getTool("pyzointeractivehelp")
        if not hw:
            return
        name = self.textCursor().selectedText().strip()
        if name == "":
            cursor = self.cursorForPosition(pos -
self.mapToGlobal(QtCore.QPoint(0, 0)))
            line = cursor.block().text()
            limit = cursor.positionInBlock()
            while limit < len(line) and (
                line[limit].isalnum() or line[limit] in (".", "_")
            ):
                limit += 1
                cursor.movePosition(cursor.Right)
            _, tokens = self.getTokensUpToCursor(cursor)
            nameBefore, name = parseLine_autocomplete(tokens)
            if nameBefore:
                name = "%s.%s" % (nameBefore, name)
        if name != "":
            hw.setObjectName(name, True)
    def processHelp(self, name=None, showError=False, addToHist=False):
        """Show help on the given full object name.
        - called when going up/down in the autocompletion list.
        - called when double clicking a name
        """
        # uses parse_autocomplete() to find baseName and objectName
        # Get help tool
        hw = pyzo.toolManager.getTool("pyzointeractivehelp")
        ass = pyzo.toolManager.getTool("pyzoassistant")
        # Get the shell
        shell = pyzo.shells.getCurrentShell()
        # Both should exist
        if not hw or not shell:
            return
        if not name:
            # Obtain name from current cursor position
            # Is this valid python?
            if self._isValidPython():
                # Obtain line from text
                cursor = self.textCursor()
                line = cursor.block().text()
                text = line[: cursor.positionInBlock()]
                # Obtain
                nameBefore, name = parseLine_autocomplete(text)
                if nameBefore:
                    name = "%s.%s" % (nameBefore, name)
        if name:
            hw.helpFromCompletion(name, addToHist)
        if ass:
            ass.showHelpForTerm(name)
    ## Callbacks
    def updateHelp(self, name):
        """A name has been highlighted, show help on that name"""
        if self._autoCompBuffer_name:
            name = self._autoCompBuffer_name + "." + name
        elif not self.completer().completionPrefix():
            # Dont update help if there is no dot or prefix;
            # the choice would be arbitrary
            return
        # Apply
        self.processHelp(name, True)
    @staticmethod
    def restoreHelp():
        hw = pyzo.toolManager.getTool("pyzointeractivehelp")
        if hw:
            hw.restoreCurrent()
    def event(self, event):
        """event(event)
        Overload main event handler so we can pass Ctrl-C Ctr-v etc, to the main
        window.
        """
        if isinstance(event, QtGui.QKeyEvent):
            # Ignore CTRL+{A-Z} since those keys are handled through the menu
            if (
                (event.modifiers() & QtCore.Qt.ControlModifier)
                and (event.key() >= QtCore.Qt.Key_A)
                and (event.key() <= QtCore.Qt.Key_Z)
            ):
                event.ignore()
                return False
        # Default behavior
        codeeditor.CodeEditor.event(self, event)
        return True
    def keyPressEvent(self, event):
        """Receive qt key event.
        From here we'l dispatch the event to perform autocompletion
        or other stuff...
        """
        # Get ordinal key
        ordKey = -1
        if event.text():
            ordKey = ord(event.text()[0])
        # Cancel any introspection in progress
        self._delayTimer._line = ""
        # Invoke autocomplete via tab key?
        if event.key() == QtCore.Qt.Key_Tab and not self.autocompleteActive():
            if pyzo.config.settings.autoComplete:
                cursor = self.textCursor()
                if cursor.position() == cursor.anchor():
                    text = cursor.block().text()[: cursor.positionInBlock()]
                    if text and (text[-1] in (Tokens.ALPHANUM + "._")):
                        self.introspect(True, False)
                        return
        super().keyPressEvent(event)
        # Analyse character/key to determine what introspection to fire
        if ordKey:
            if (
                ordKey >= 48 or ordKey in [8, 46]
            ) and pyzo.config.settings.autoComplete == 1:
                # If a char that allows completion or backspace or dot was
pressed
                self.introspect(True)
            elif ordKey >= 32:
                # Printable chars, only calltip
                self.introspect()
        elif event.key() in [QtCore.Qt.Key_Left, QtCore.Qt.Key_Right]:
            self.introspect()
class CallTipObject:
    """Object to help the process of call tips.
    An instance of this class is created for each call tip action.
    """
    def __init__(self, textCtrl, name, offset):
        self.textCtrl = textCtrl
        self.name = name
        self.bufferName = name
        self.offset = offset
    def tryUsingBuffer(self):
        """tryUsingBuffer()
        Try performing this callTip using the buffer.
        Returns True on success.
        """
        bufferName = self.textCtrl._callTipBuffer_name
        t = time.time() - self.textCtrl._callTipBuffer_time
        if self.bufferName == bufferName and t < 0:
            self._finish(self.textCtrl._callTipBuffer_result)
            return True
        else:
            return False
    def finish(self, callTipText):
        """finish(callTipText)
        Finish the introspection using the given calltipText.
        Will also automatically call setBuffer.
        """
        self.setBuffer(callTipText)
        self._finish(callTipText)
    def setBuffer(self, callTipText, timeout=4):
        """setBuffer(callTipText)
        Sets the buffer with the provided text."""
        self.textCtrl._callTipBuffer_name = self.bufferName
        self.textCtrl._callTipBuffer_time = time.time() + timeout
        self.textCtrl._callTipBuffer_result = callTipText
    def _finish(self, callTipText):
        self.textCtrl.calltipShow(self.offset, callTipText, True)
class AutoCompObject:
    """Object to help the process of auto completion.
    An instance of this class is created for each auto completion action.
    """
    def __init__(self, textCtrl, name, needle):
        self.textCtrl = textCtrl
        self.bufferName = name  # name to identify with
        self.name = name  # object to find attributes of
        self.needle = needle  # partial name to look for
        self.names = set()  # the names (use a set to prevent duplicates)
        self.importNames = []
        self.importLines = {}
    def addNames(self, names):
        """addNames(names)
        Add a list of names to the collection.
        Duplicates are removed."""
        self.names.update(names)
    def tryUsingBuffer(self):
        """tryUsingBuffer()
        Try performing this auto-completion using the buffer.
        Returns True on success.
        """
        bufferName = self.textCtrl._autoCompBuffer_name
        t = time.time() - self.textCtrl._autoCompBuffer_time
        if self.bufferName == bufferName and t < 0:
            self._finish(self.textCtrl._autoCompBuffer_result)
            return True
        else:
            return False
    def finish(self):
        """finish()
        Finish the introspection using the collected names.
        Will automatically call setBuffer.
        """
        # Remember at the object that started this introspection
        # and get sorted names
        names = self.setBuffer(self.names)
        # really finish
        self._finish(names)
    def setBuffer(self, names=None, timeout=None):
        """setBuffer(names=None)
        Sets the buffer with the provided names (or the collected names).
        Also returns a list with the sorted names."""
        # Determine timeout
        # Global namespaces change more often than local one, plus when
        # typing a xxx.yyy, the autocompletion buffer changes and is thus
        # automatically refreshed.
        # I've once encountered a wrong autocomp list on an object, but
        # haven' been able to reproduce it. It was probably some odity.
        if timeout is None:
            if self.bufferName:
                timeout = 5
            else:
                timeout = 1
        # Get names
        if names is None:
            names = self.names
        # Make list and sort
        names = list(names)
        names.sort(key=str.upper)
        # Store
        self.textCtrl._autoCompBuffer_name = self.bufferName
        self.textCtrl._autoCompBuffer_time = time.time() + timeout
        self.textCtrl._autoCompBuffer_result = names
        # Return sorted list
        return names
    def _finish(self, names):
        # Show completion list if required.
        self.textCtrl.autocompleteShow(len(self.needle), names, self.name != "")
    def nameInImportNames(self, importNames):
        """nameInImportNames(importNames)
        Test whether the name, or a base part of it is present in the
        given list of names. Returns the (part of) the name that's in
        the list, or None otherwise.
        """
        baseName = self.name
        while baseName not in importNames:
            if "." in baseName:
                baseName = baseName.rsplit(".", 1)[0]
            else:
                baseName = None
                break
        return baseName
if __name__ == "__main__":
    app = QtWidgets.QApplication([])
    win = BaseTextCtrl(None)
    #     win.setStyle('.py')
    tmp = "foo(bar)\nfor bar in range(5):\n  print bar\n"
    tmp += "\nclass aap:\n  def monkey(self):\n    pass\n\n"
    tmp += "a\u20acb\n"
    win.setPlainText(tmp)
    win.show()
    app.exec_()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module codeparser
Analyses the source code to get the structure of a module/script.
This can be used for fictive introspection, and to display the
structure of a source file in for example a tree widget.
"""
# TODO: replace this module, get data from the syntax highlighter in the code
editor
import time, threading, re
import pyzo
# Define regular expression patterns
classPattern = r"^\s*"  # Optional whitespace
classPattern += r"(cp?def\s+)?"  # Cython preamble + whitespace
classPattern += r"class\s+"  # The class keyword + whitespace
classPattern += r"([a-zA-Z_][a-zA-Z_0-9]*)\s*"  # The NAME + optional whitespace
classPattern += r"(\(.*?\))?"  # The superclass(es)
classPattern += r"\s*:"  # Optional whitespace and the colon
#
defPattern = r"^\s*"  # Optional whitespace
defPattern += r"(async )?"  # Optional async keyword
defPattern += r"(cp?)?def\s+"  # The Cython preamble, def keyword and whitespace
defPattern += r"([a-zA-Z_][\*a-zA-Z_0-9]*\s+)?"  # Optional Cython return type
defPattern += r"([a-zA-Z_][a-zA-Z_0-9]*)\s*"  # The NAME + optional whitespace
defPattern += r"\((.*?)\)"  # The SIGNATURE
# defPattern += r'\s*:' # Optional whitespace and the colon
# Leave the colon, easier for cython
class Job:
    """Simple class to represent a job."""
    def __init__(self, text, editorId):
        self.text = text
        self.editorId = editorId
class Result:
    """Simple class to represent a parser result."""
    def __init__(self, rootItem, importList, editorId):
        self.rootItem = rootItem
        self.importList = importList
        self.editorId = editorId
    def isMatch(self, editorId):
        """isMatch(editorId):
```

```
        Returns whether the result matches with the given editorId.
        The editorId can also be an editor instance."""
        if isinstance(editorId, int):
            return self.editorId == editorId
        else:
            return self.editorId == id(editorId)
class Parser(threading.Thread):
    """Parser
    Parsing sourcecode in a separate thread, this class obtains
    introspection informarion. This class is also the interface
    to the parsed information; it has methods that can be used
    to extract information from the result.
    """
    def __init__(self):
        threading.Thread.__init__(self)
        # Reference current job
        self._job = None
        # Reference to last result
        self._result = None
        # Lock to enable save threading
        self._lock = threading.RLock()
        # Set deamon
        self.daemon = True
        self._exit = False
    def stop(self, timeout=1.0):
        self._exit = True
        self.join(timeout)
    def parseThis(self, editor):
        """parseThis(editor)
        Give the parser new text to parse.
        If the parser is busy parsing text, it will stop doing that
        and start anew with the most recent version of the text.
        """
        # Get text
        text = editor.toPlainText()
        # Make job
        self._lock.acquire()
        self._job = Job(text, id(editor))
        self._lock.release()
    def getFictiveNameSpace(self, editor):
        """getFictiveNameSpace(editor)
        Produce the fictive namespace, based on the current position.
        A list of names is returned.
        """
        # Obtain result
        result = self._getResult()
        if result is None or not result.isMatch(editor):
            return []
        # Get linenr and indent. These are used to establish the namespace
        # based on indentation.
        cursor = editor.textCursor()
        linenr = cursor.blockNumber()
        index = cursor.positionInBlock()
        # init empty namespace and item list
        namespace = []
        items = result.rootItem.children
        curIsClass = False  # to not add methods (of classes)
        while items:
            curitem = None
            for item in items:
                # append name
                if not curIsClass and item.type in ["class", "def"]:
                    namespace.append(item.name)
                # check if this is the one only last one remains
                if (
                    item.type in ["class", "def"]
                    and item.linenr <= linenr
                    and item.linenr2 > linenr
                ):
                    curitem = item
            # prepare for next round
            if curitem and curitem.indent < index:
                items = curitem.children
                if curitem.type == "class":
                    curIsClass = True
                else:
                    curIsClass = False
            else:
                items = []
        return namespace
    def getFictiveClass(self, name, editor, handleSelf=False):
        """getFictiveClass(name, editor, handleSelf=False)
        Return the fictive class object of the given name, or None
        if it does not exist. If handleSelf is True, automatically
        handles "self." names.
        """
        return self._getFictiveItem(name, "class", editor, handleSelf)
    def getFictiveSignature(self, name, editor, handleSelf=False):
        """getFictiveSignature(name, editor, handleSelf=False)
        Get the signature of the fictive function or method of the
        given name. Returns None if the given name is not a known
        function or method. If handleSelf is True, automatically
        handles "self." names.
        """
        # Get item being a function
        item = self._getFictiveItem(name, "def", editor, handleSelf)
        # Get item being a class
        if not item:
            item = self._getFictiveItem(name, "class", editor, handleSelf)
            if item:
                for subItem in item.children:
                    if subItem.name == "__init__" and subItem.type == "def":
                        item = subItem
                        break
                else:
                    item = None
        # Process or return None if there was no item
        if item:
            nameParts = name.split(".")
            return "{}({})".format(nameParts[-1], item.sig)
        else:
            return None
    def getFictiveImports(self, editor):
        """getFictiveImports(editor)
        Get the fictive imports of this source file.
        tuple:
        - list of names that are imported,
        - a dict with the line to import each name
        """
        # Obtain result
        result = self._getResult()
        if result is None or not result.isMatch(editor):
            return [], []
        # Extract list of names and dict of lines
        imports = []
        importlines = {}
        for item in result.importList:
            imports.append(item.name)
            importlines[item.name] = item.text
        return imports, importlines
    def _getResult(self):
        """getResult()
        Savely Obtain result.
        """
        self._lock.acquire()
        result = self._result
        self._lock.release()
        return result
    def _getFictiveItem(self, name, type, editor, handleSelf=False):
        """_getFictiveItem(name, type, editor, handleSelf=False)
        Obtain the fictive item of the given name and type.
        If handleSelf is True, will handle "self." correctly.
        Intended for internal use.
        """
        # Obtain result
        result = self._getResult()
        if result is None or not result.isMatch(editor):
            return None
        # Split name in parts
        nameParts = name.split(".")
        # Try if the first part represents a class instance
        if handleSelf:
            item = self._getFictiveCurrentClass(editor, nameParts[0])
            if item:
                nameParts[0] = item.name
        # Init
        name = nameParts.pop(0)
        items = result.rootItem.children
        theItem = None
        # Search for name
        while items:
            for item in items:
                if item.name == name:
                    # Found it
                    if nameParts:
                        # Go down one level
                        name = nameParts.pop(0)
                        items = item.children
                        break
                    else:
                        # This is it, is it what we wanted?
                        if item.type == type:
```

```
                        theItem = item
                        items = []
                        break
            else:
                # Did not find it
                items = []
        return theItem
    def _getFictiveCurrentClass(self, editor, selfname):
        """_getFictiveCurrentClass(editor, selfname)
        Get the fictive object for the class referenced
        using selfname (usually 'self').
        Intendef for internal use.
        """
        # Obtain result
        result = self._getResult()
        if result is None:
            return None
        # Get linenr and indent
        cursor = editor.textCursor()
        linenr = cursor.blockNumber()
        index = cursor.positionInBlock()
        # Init
        items = result.rootItem.children
        theclass = None
        while items:
            curitem = None
            for item in items:
                # check if this is the one only last one remains
                if item.linenr <= linenr:
                    if not item.linenr2 > linenr:
                        continue
                    curitem = item
                    if item.type == "def" and item.selfname == selfname:
                        theclass = item.parent
                else:
                    break
            # prepare for next round
            if curitem and curitem.indent < index:
                items = curitem.children
            else:
                items = []
        # return
        return theclass
    def run(self):
        """run()
        This is the main loop.
        """
        time.sleep(0.5)
        try:
            while True:
                time.sleep(0.1)
                if self._exit:
                    return
                if self._job:
                    # Savely obtain job
                    self._lock.acquire()
                    job = self._job
                    self._job = None
                    self._lock.release()
                    # Analyse job
                    result = self._analyze(job)
                    # Savely store result
                    self._lock.acquire()
                    self._result = result
                    self._lock.release()
                    # Notify
                    if pyzo.editors is not None:
                        pyzo.editors.parserDone.emit()
        except AttributeError:
            pass  # when python exits, time can be None...
    def _analyze(self, job):
        """The core function.
        Analyses the source code.
        Produces:
        - a tree of FictiveObject objects.
        - a (flat) list of the same object
        - a list of imports
        """
        # Remove multiline strings
        text = washMultilineStrings(job.text)
        # Split text in lines
        lines = text.splitlines()
        lines.insert(0, "")  # so the lines start at 1
        # The structure object. It will first only consist of class and defs
        # the rest will be inserted afterwards.
        root = FictiveObject("root", 0, -1, "root")
        # Also keep a flat list (while running this function)
        flatList = []
        # Cells and imports are inserted in the structure afterwards
        leafs = []
        # Keep a list of imports
        importList = []
        # To know when to make something new when for instance a class is
defined
        # in an if statement, we keep track of the last valid node/object:
        # Put inside a list, so we can set it from inside a subfuncion
        lastObject = [root]
        # Define funcion to put an item in the structure in the right parent
        def appendToStructure(object):
            # find position in structure to insert
            node = lastObject[0]
            while (object.indent <= node.indent) and (node is not root):
                node = node.parent
            # insert object
            flatList.append(object)
            node.children.append(object)
            object.parent = node
            lastObject[0] = object
        # Find objects!
        # type can be: cell, class, def, import, var
        for i in range(len(lines)):
            # Obtain line
            line = lines[i]
            # Should we stop?
            if self._job or self._exit:
                break
            # Remove indentation
            tmp = line.lstrip()
            indent = len(line) - len(tmp)
            line = tmp.rstrip()
            # Detect cells
            if (
                line.startswith("##")
                or line.startswith("#%%")
                or line.startswith("# %%")
            ):
                if line.startswith("##"):
                    name = line[2:].lstrip()
                elif line.startswith("#%%"):
                    name = line[3:].lstrip()
                else:
                    name = line[4:].lstrip()
                item = FictiveObject("cell", i, indent, name)
                leafs.append(item)
                # Next! (we have to put this before the elif stuff below
                # because it looks like a comment!)
                continue
            # Split in line and comment
            line, tmp, cmnt = line.partition("#")
            line, cmnt = line.rstrip(), cmnt.lower().strip()
            # Detect todos
            if cmnt and (cmnt.startswith("todo:") or cmnt.startswith("2do:")):
                item = FictiveObject("todo", i, indent, cmnt)
                item.linenr2 = i + 1  # a todo is active at one line only
                leafs.append(item)
            # Continue of no line left
            if not line:
                continue
            # Find last valid node. As the indent of the root is set to -1,
            # this will always stop at the root
            while indent <= lastObject[0].indent:
                lastObject[0].linenr2 = i  # close object
                lastObject[0] = lastObject[0].parent
            # Make a lowercase version of the line
            foundSomething = False
            # Detect classes
            if not foundSomething:
                classResult = re.search(classPattern, line)
                if classResult:
                    foundSomething = True
                    # Get name
                    name = classResult.group(2)
                    item = FictiveObject("class", i, indent, name)
                    appendToStructure(item)
                    item.supers = []
                    item.members = []
                    # Get inheritance
                    supers = classResult.group(3)
                    if supers:
                        supers = supers[1:-1].split(",")
                        supers = [tmp.strip() for tmp in supers]
                        item.supers = [tmp for tmp in supers if tmp]
            # Detect functions and methods (also multiline)
            if (not foundSomething) and line.count("def "):
                # Get a multiline version (for long defs)
                multiLine = line
                for ii in range(1, 16):
                    if i + ii < len(lines):
                        multiLine += " " + lines[i + ii].strip()
                # Get result
                defResult = re.search(defPattern, multiLine)
                if defResult:
                    # Get name
                    name = defResult.group(4)
                    item = FictiveObject("def", i, indent, name)
                    appendToStructure(item)
                    item.selfname = None  # will be filled in if a valid method
                    item.sig = defResult.group(5)
                    # is it a method? -> add method to attr and find selfname
                    if item.parent.type == "class":
                        item.parent.members.append(name)
                        # Find what is used as "self"
                        i2 = line.find("(")
                        i4 = line.find(",", i2)
                        if i4 < 0:
                            i4 = line.find(")", i2)
                        if i4 < 0:
                            i4 = i2
                        selfname = line[i2 + 1 : i4].strip()
                        if selfname:
                            item.selfname = selfname
            elif line.count("import "):
                if line.startswith("import "):
                    for name in ParseImport(line[7:]):
                        item = FictiveObject("import", i, indent, name)
                        item.text = line
                        item.linenr2 = i + 1  # an import is active at one line
only
                        leafs.append(item)
                        importList.append(item)
                elif line.startswith("from "):
                    i1 = line.find(" import ")
                    for name in ParseImport(line[i1 + 8 :]):
                        if not IsValidName(name):
                            continue  # we cannot do that!
                        item = FictiveObject("import", i, indent, name)
                        item.text = line
                        item.linenr2 = i + 1  # an import is active at one line
only
                        leafs.append(item)
                        importList.append(item)
            elif (
                not indent and line.startswith("if __name__ ==") and "__main__"
in line
            ):
                item = FictiveObject("nameismain", i, indent, "__main__")
                item.text = line
                appendToStructure(item)
            elif line.count("="):
                if lastObject[0].type == "def" and lastObject[0].selfname:
                    selfname = lastObject[0].selfname + "."
                    line = line.partition("=")[0]
                    if line.count(selfname):
                        # A lot of ifs here. If we got here, the line is part of
                        # a valid method and contains the selfname before the =.
                        # Now we need to establish whether there is a valid
                        # assignment done here...
                        parts = line.split(",")  # handle tuples
                        for part in parts:
                            part = part.strip()
                            part2 = part[len(selfname) :]
                            if part.startswith(selfname) and IsValidName(part2):
                                # add to the list if not already present
                                defItem = lastObject[0]
                                classItem = lastObject[0].parent
                                #
                                item = FictiveObject("attribute", i, indent,
part2)
                                item.parent = defItem
                                defItem.children.append(item)
                                if part2 not in classItem.members:
                                    classItem.members.append(part2)
        ## Post processing
        def getTwoItems(series, linenr):
            """Return the two items just above and below the
            given linenr. The object always is a class or def.
            """
            # find object after linenr
            object1, object2 = None, None  # if no items at all
            i = -1
            for i in range(len(series)):
                object = series[i]
                if object.type not in ["class", "def"]:
                    continue
                if object.linenr > linenr:
                    object2 = object
                    break
            # find object just before linenr
            for ii in range(i, -1, -1):
                object = series[ii]
                if object.type not in ["class", "def"]:
                    continue
                if object.linenr < linenr:
                    object1 = object
                    break
            # return result
            return object1, object2
        # insert the leafs (backwards as the last inserted is at the top)
        for leaf in reversed(leafs):
            ob1, ob2 = getTwoItems(flatList, leaf.linenr)
            if ob1 is None:  # also if ob2 is None
                # insert in root
                root.children.insert(0, leaf)
                leaf.parent = root
                continue
            if ob2 is None:
                ob2parent = root
            else:
                ob2parent = ob2.parent
            # get the object IN which to insert it: ob1
            sibling = None
            while 1:
                canGoDeeper = ob1 is not ob2parent
                canGoDeeper = canGoDeeper and ob1 is not root
                shouldGoDeeper = ob1.indent >= leaf.indent
                shouldGoDeeper = shouldGoDeeper or ob1.linenr2 < leaf.linenr
                if canGoDeeper and shouldGoDeeper:
                    sibling = ob1
                    ob1 = ob1.parent
                else:
                    break
            # insert into ob1, after sibling (if available)
            L = ob1.children
            if sibling:
                i = L.index(sibling)
                L.insert(i + 1, leaf)
            else:
                L.insert(0, leaf)
        # Return result
        return Result(root, importList, job.editorId)
## Helper classes and functions
class FictiveObject:
    """An un-instantiated object.
    type can be class, def, import, cell, todo
    extra stuff:
    class   - supers, members
    def     - selfname
    imports - text
    cell    -
    todo    -
    attribute -
    """
    def __init__(self, type, linenr, indent, name):
        self.children = []
        self.type = type
        self.linenr = linenr  # at which line this object starts
        self.linenr2 = 9999999  # at which line it ends
        self.indent = indent
        self.name = name
        self.sig = ""  # for functions and methods
namechars = "abcdefghijklmnopqrstuvwxyz_0123456789"
def IsValidName(name):
    """Given a string, checks whether it is a
    valid name (dots are not valid!)
    """
    if not name:
        return False
    name = name.lower()
    if name[0] not in namechars[0:-10]:
        return False
    tmp = map(lambda x: x not in namechars, name[2:])
    return sum(tmp) == 0
def ParseImport(names):
    for part in names.split(","):
        i1 = part.find(" as ")
        if i1 > 0:
            name = part[i1 + 3 :].strip()
        else:
            name = part.strip()
        yield name
def findString(text, s, i):
    """findString(text, s)
    Find s in text, but only if s is not in a string or commented
    Helper function for washMultilineStrings"""
    while True:
        i = _findString(text, s, i)
        if i < -1:
            i = -i + 1
        else:
            break
    return i
def _findString(text, s, i):
    """Helper function of findString, which is called recursively
    until a match is found, or it is clear there is no match."""
    # Find occurrence
    i2 = text.find(s, i)
    if i2 < 0:
        return -1
    # Find newline  (if none, we're done)
    i1 = text.rfind("\n", 0, i2)
    if i1 < 0:
        return i2
    # Extract the part on the line up to the match
    line = text[i1:i2]
    # Count quotes, we're done if we found none
    if not line.count('"') and not line.count("'") and not line.count("#"):
        return i2
    # So we found quotes, now really count them ...
    prev = ""
    inString = ""  # this is a boolean combined with a flag which quote was used
    isComment = False
    for c in line:
        if c == "#":
            if not inString:
                isComment = True
                break
        elif c in "\"'":
            if not inString:
                inString = c
            elif prev != "\\":
                if inString == c:
                    inString = ""  # exit string
                else:
                    pass  # the other quote can savely be used inside this 
string
        prev = c
    # If we are in a string, this match is false ...
    if inString or isComment:
        return -i2  # indicate failure and where to continue
    else:
        return i2  # all's right
def washMultilineStrings(text):
    """washMultilineStrings(text)
    Replace all text within multiline strings with dummy chars
    so that it is not parsed.
    """
    i = 0
    s1 = "'''"
    s2 = '"""'
    while i < len(text):
        # Detect start of a multiline comment (there are two versions)
        i1 = findString(text, s1, i)
        i2 = findString(text, s2, i)
        # Stop if nothing found ...
        if i1 == -1 and i2 == -1:
            break
        else:
            # Make no result be very large
            if i1 == -1:
                i1 = 2**60
            if i2 == -1:
                i2 = 2**60
            # Find end of the multiline comment
            if i1 < i2:
                i3 = i1 + 3
                i4 = text.find(s1, i3)
            else:
                i3 = i2 + 3
                i4 = text.find(s2, i3)
            # No end found -> take all text, unclosed string!
            if i4 == -1:
                i4 = 2**32
            # Leave only the first two quotes of the start of the comment
            i3 -= 1
            i4 += 3
            # Replace all non-newline chars
            tmp = re.sub(r"\S", " ", text[i3:i4])
            text = text[:i3] + tmp + text[i3 + len(tmp) :]
            # Prepare for next round
            i = i4 + 1
    return text
"""
## testing skipping of multiline strings
def ThisShouldNotBeVisible():
  pass
class ThisShouldNotBeVisibleEither():
  pass
"""
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2014, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module to deal with command line arguments.
In specific, this allows doing "pyzo some_file.py" and the file will be
opened in an existing pyzo window (if available) or a new pyzo process
is started to open the file.
This module is used at the very early stages of starting pyzo, and also
in main.py to apply any command line args for the current process, and
to close down the server when pyzo is closed.
"""
import sys
import os
from yoton.clientserver import RequestServer, do_request
import pyzo
# Local address to host on. we use yoton's port hash to have an arbitrary port
ADDRESS = "localhost:pyzoserver"
class Server(RequestServer):
    """Server that listens on a port for commands.
    The commands can be send by executing the Pyzo executable with
    command line arguments.
    """
    def handle_request(self, request):
        """This is where the requests enter."""
        # Get command
        request = request.strip()
        command, _, arg = request.partition(" ")
        # Handle command
        try:
            reply = handle_command(command, arg)
        except Exception as err:
            msg = "Error handling request %r:\n%s" % (request, str(err))
            pyzo.callLater(print, msg)
            return msg
        else:
            pyzo.callLater(print, "Request:", request)
            pyzo.callLater(print, "Reply:", reply)
            return reply
def handle_command(command, arg):
    """Function that handles all pyzo commands.
    This gets called either from the server, or from the code that
    processed command line args.
    """
    if not command:
        return "empty command?"
    elif command == "testerr":
        return 1 / 0
    elif command == "stopserver":
        # For efficiently stopping the server
        if server:
            server.stop()
            return "Stopped the server"
    elif command == "echo":
        # For testing
        return "echo %r" % arg
    elif command == "open":
        # Open a file in the editor
        if not arg:
            return "The open command requires a filename."
        pyzo.callLater(pyzo.editors.loadFile, arg)
        return "Opened file %r" % arg
    elif command == "new":
        # Open a new (temp) file in the editor
        pyzo.callLater(pyzo.editors.newFile)
        return "Created new file"
    elif command == "close":
        # Close pyzo
        pyzo.callLater(pyzo.main.close)
        return "Closing Pyzo"
    else:
        # Assume the user wanted to open a file
        fname = (command + " " + arg).rstrip()
        if not pyzo.editors:
            return "Still warming up ..."
        else:
            pyzo.callLater(pyzo.editors.loadFile, fname)
            return "Try opening file %r" % fname
    # We should always return. So if we get here, it is a bug.
    # Return something so that we can be aware.
    return "error " + command
def handle_cmd_args():
    """Handle command line arguments by sending them to the server.
    Returns a result string if any commands were processed, and None
    otherwise.
    """
    args = sys.argv[1:]
    request = " ".join(arg for arg in args if not arg.startswith("--"))
    if "psn_" in request and not os.path.isfile(request):
        request = " ".join(args[1:])  # An OSX thing when clicking app icon
    request = request.strip()
    #
    if not request:
        return None
    else:
        # Always send to server, even if we are the ones that run the server
        try:
            return do_request(ADDRESS, request, 0.4).rstrip()
        except Exception as err:
            print("Could not process command line args:\n%s" % str(err))
            return None
def stop_our_server():
    """Stop our server, for shutting down nicely.
    This is faster than calling server.stop(), because in the latter
    case the server will need to timeout (0.25 s) before it sees that
    it needs to stop.
    """
    if is_our_server_running():
        try:
            server.stop()  # Post a stop message
            do_request(ADDRESS, "stopserver", 0.1)  # trigger
            print("Stopped our command server.")
        except Exception as err:
            print("Failed to stop command server:")
            print(err)
def is_our_server_running():
    """Return True if our server is running. If it is, this process
    is the main Pyzo; the first Pyzo that was started. If the server is
    not running, this is probably not the first Pyzo, but there might
    also be problem with starting the server.
    """
    return server and server.is_alive()
def is_pyzo_server_running():
    """Test whether the Pyzo server is running *somewhere* (not
    necesarily in this process).
    """
    try:
        res = do_request(ADDRESS, "echo", 0.2)
        return res.startswith("echo")
    except Exception:
        return False
# Shold we start the server?
_try_start_server = True
if sys.platform.startswith("win"):
    _try_start_server = not is_pyzo_server_running()
# Create server
server_err = None
server = None
try:
    if _try_start_server:
        server = Server(ADDRESS)
        server.start()
except OSError as err:
    server_err = err
    server = None
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" compact tab widget class
See docs of the tab widget.
"""
from pyzo.qt import QtCore, QtGui, QtWidgets  # noqa
import sys
if sys.version_info[0] < 3:
    str = unicode  # noqa
    ELLIPSIS = unichr(8230)  # noqa
else:
    ELLIPSIS = chr(8230)
# Constants for the alignments of tabs
MIN_NAME_WIDTH = 4
MAX_NAME_WIDTH = 64
## Define style sheet for the tabs
STYLESHEET = """
QTabWidget::pane { /* The tab widget frame */
    border-top: 0px solid #A09B90;
}
QTabWidget::tab-bar {
    left: 0px; /* move to the right by x px */
}
/* Style the tab using the tab sub-control. Note that
 it reads QTabBar _not_ QTabWidget */
QTabBar::tab {
    background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
                stop: 0.0 rgba(220,220,220,128),
                stop: 0.4 rgba(200,200,200,128),
                stop: 1.0 rgba(100,100,100,128) );
    border: 1px solid #A09B90;
    border-bottom-color: #DAD5CC; /* same as the pane color */
    border-top-left-radius: 4px;
    border-top-right-radius: 4px;
    min-width: 5ex;
    padding-bottom: PADDING_BOTTOMpx;
    padding-top: PADDING_TOPpx;
    padding-left: PADDING_LEFTpx;
    padding-right: PADDING_RIGHTpx;
    margin-right: -1px; /* "combine" borders */
}
QTabBar::tab:last {
    margin-right: 0px;
}
/* Style the selected tab, hoovered tab, and other tabs. */
QTabBar::tab:hover {
    background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
                stop: 0.0 rgba(245,250,255,128),
                stop: 0.4 rgba(210,210,210,128),
                stop: 1.0 rgba(200,200,200,128) );
}
QTabBar::tab:selected {
    background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
                stop: 0.0 rgba(0,0,128,128),
                stop: 0.12 rgba(0,0,128,128),
                stop: 0.120001 rgba(245,250,255,128),
                stop: 0.4 rgba(210,210,210,128),
                stop: 1.0 rgba(200,200,200,128) );
}
QTabBar::tab:selected {
    border-width: 1px;
    border-bottom-width: 0px;
    border-top-left-radius: 5px;
    border-top-right-radius: 5px;
    border-color: #333;
}
QTabBar::tab:!selected {
    margin-top: 3px; /* make non-selected tabs look smaller */
}
"""
## Define tab widget class
class TabData:
    """To keep track of real names of the tabs, but also keep supporting
    tabData.
    """
    def __init__(self, name):
        self.name = name
        self.data = None
class CompactTabBar(QtWidgets.QTabBar):
    """CompactTabBar(parent, *args, padding=(4,4,6,6), preventEqualTexts=True)
    Tab bar corresponcing to the CompactTabWidget.
    With the "padding" argument the padding of the tabs can be chosen.
    It should be an integer, or a 4 element tuple specifying the padding
    for top, bottom, left, right. When a tab has a button,
    the padding is the space between button and text.
    With preventEqualTexts to True, will reduce the amount of eliding if
    two tabs have (partly) the same name, so that they can always be
    distinguished.
    """
    # Add signal to be notified of double clicks on tabs
    tabDoubleClicked = QtCore.Signal(int)
    barDoubleClicked = QtCore.Signal()
    def __init__(self, *args, padding=(4, 4, 6, 6), preventEqualTexts=True):
        QtWidgets.QTabBar.__init__(self, *args)
        # Put tab widget in document mode
        self.setDocumentMode(True)
        # Widget needs to draw its background (otherwise Mac has a dark bg)
        self.setDrawBase(False)
        if sys.platform == "darwin":
            self.setAutoFillBackground(True)
        # Set whether we want to prevent eliding for names that start the same.
        self._preventEqualTexts = preventEqualTexts
        # Allow moving tabs around
        self.setMovable(True)
        # Get padding
        if isinstance(padding, (int, float)):
            padding = padding, padding, padding, padding
        elif isinstance(padding, (tuple, list)):
            pass
        else:
            raise ValueError("Invalid value for padding.")
        # Set style sheet
        stylesheet = STYLESHEET
        stylesheet = stylesheet.replace("PADDING_TOP", str(padding[0]))
        stylesheet = stylesheet.replace("PADDING_BOTTOM", str(padding[1]))
        stylesheet = stylesheet.replace("PADDING_LEFT", str(padding[2]))
        stylesheet = stylesheet.replace("PADDING_RIGHT", str(padding[3]))
        self.setStyleSheet(stylesheet)
        # We do our own eliding
        self.setElideMode(QtCore.Qt.ElideNone)
        # Make tabs wider if there's plenty space?
        self.setExpanding(False)
        # If there's not enough space, use scroll buttons
        self.setUsesScrollButtons(True)
        # When a tab is removed, select previous
        self.setSelectionBehaviorOnRemove(self.SelectPreviousTab)
        # Init alignment parameters
        self._alignWidth = MIN_NAME_WIDTH  # Width in characters
        self._alignWidthIsReducing = False  # Whether in process of reducing
        # Create timer for aligning
        self._alignTimer = QtCore.QTimer(self)
        self._alignTimer.setInterval(10)
        self._alignTimer.setSingleShot(True)
        self._alignTimer.timeout.connect(self._alignRecursive)
    def _compactTabBarData(self, i):
        """_compactTabBarData(i)
        Get the underlying tab data for tab i. Only for internal use.
        """
        # Get current TabData instance
        tabData = QtWidgets.QTabBar.tabData(self, i)
        if (tabData is not None) and hasattr(tabData, "toPyObject"):
            tabData = tabData.toPyObject()  # Older version of Qt
        # If none, make it as good as we can
        if not tabData:
            name = str(QtWidgets.QTabBar.tabText(self, i))
            tabData = TabData(name)
            QtWidgets.QTabBar.setTabData(self, i, tabData)
        # Done
        return tabData
    ## Overload a few methods
    def mouseDoubleClickEvent(self, event):
        i = self.tabAt(event.pos())
        if i == -1:
            # There was no tab under the cursor
            self.barDoubleClicked.emit()
        else:
            # Tab was double clicked
            self.tabDoubleClicked.emit(i)
    def mousePressEvent(self, event):
        if event.button() == QtCore.Qt.MiddleButton:
            i = self.tabAt(event.pos())
            if i >= 0:
                self.parent().tabCloseRequested.emit(i)
                return
        super().mousePressEvent(event)
    def setTabData(self, i, data):
        """setTabData(i, data)
        Set the given object at the tab with index 1.
        """
        # Get underlying python instance
        tabData = self._compactTabBarData(i)
        # Attach given data
        tabData.data = data
    def tabData(self, i):
        """tabData(i)
        Get the tab data at item i. Always returns a Python object.
        """
        # Get underlying python instance
        tabData = self._compactTabBarData(i)
        # Return stored data
        return tabData.data
    def setTabText(self, i, text):
        """setTabText(i, text)
        Set the text for tab i.
        """
        tabData = self._compactTabBarData(i)
        if text != tabData.name:
            tabData.name = text
            self.alignTabs()
    def tabText(self, i):
        """tabText(i)
        Get the title of the tab at index i.
        """
        tabData = self._compactTabBarData(i)
        return tabData.name
    ## Overload events and protected functions
    def tabInserted(self, i):
        QtWidgets.QTabBar.tabInserted(self, i)
        # Is called when a tab is inserted
        # Get given name and store
        name = str(QtWidgets.QTabBar.tabText(self, i))
        tabData = TabData(name)
        QtWidgets.QTabBar.setTabData(self, i, tabData)
        # Update
        self.alignTabs()
    def tabRemoved(self, i):
        QtWidgets.QTabBar.tabRemoved(self, i)
        # Update
        self.alignTabs()
    def resizeEvent(self, event):
        QtWidgets.QTabBar.resizeEvent(self, event)
        self.alignTabs()
    def showEvent(self, event):
        QtWidgets.QTabBar.showEvent(self, event)
        self.alignTabs()
    ## For aligning
    def alignTabs(self):
        """alignTabs()
        Align the tab items. Their names are ellided if required so that
        all tabs fit on the tab bar if possible. When there is too little
        space, the QTabBar will kick in and draw scroll arrows.
        """
        # Set name widths correct (in case new names were added)
        self._setMaxWidthOfAllItems()
        # Start alignment process
        self._alignWidthIsReducing = False
        self._alignTimer.start()
    def _alignRecursive(self):
        """_alignRecursive()
        Recursive alignment of the items. The alignment process
        should be initiated from alignTabs().
        """
        # Only if visible
        if not self.isVisible():
            return
        # Get tab bar and number of items
        N = self.count()
        # Get right edge of last tab and left edge of corner widget
        pos1 = self.tabRect(0).topLeft()
        pos2 = self.tabRect(N - 1).topRight()
        cornerWidget = self.parent().cornerWidget()
        if cornerWidget:
            pos3 = cornerWidget.pos()
        else:
            pos3 = QtCore.QPoint(int(self.width()), 0)
        x1 = pos1.x()
        x2 = pos2.x()
        x3 = pos3.x()
        alignMargin = x3 - (x2 - x1) - 3  # Must be positive (has margin)
        # Are the tabs too wide?
        if alignMargin < 0:
            # Tabs extend beyond corner widget
            # Reduce width then
            self._alignWidth -= 1
            self._alignWidth = max(self._alignWidth, MIN_NAME_WIDTH)
            # Apply
            self._setMaxWidthOfAllItems()
            self._alignWidthIsReducing = True
            # Try again if there's still room for reduction
            if self._alignWidth > MIN_NAME_WIDTH:
                self._alignTimer.start()
        elif alignMargin > 10 and not self._alignWidthIsReducing:
            # Gap between tabs and corner widget is a bit large
            # Increase width then
            self._alignWidth += 1
            self._alignWidth = min(self._alignWidth, MAX_NAME_WIDTH)
            # Apply
            itemsElided = self._setMaxWidthOfAllItems()
            # Try again if there's still room for increment
            if itemsElided and self._alignWidth < MAX_NAME_WIDTH:
                self._alignTimer.start()
                # self._alignTimer.timeout.emit()
        else:
            pass  # margin is good
    def _getAllNames(self):
        """_getAllNames()
        Get a list of all (full) tab names.
        """
        return [self._compactTabBarData(i).name for i in range(self.count())]
    def _setMaxWidthOfAllItems(self):
        """_setMaxWidthOfAllItems()
        Sets the maximum width of all items now, by eliding the names.
        Returns whether any items were elided.
        """
        # Get whether an item was reduced in size
        itemReduced = False
        for i in range(self.count()):
            # Get width
            w = self._alignWidth
            # Get name
            name = self._compactTabBarData(i).name
            # If its too long, first make it shorter by stripping dir names
            if (w + 1) < len(name) and "/" in name:
                name = name.split("/")[-1]
            # Check if we can reduce the name size, correct w if necessary
            if ((w + 1) < len(name)) and self._preventEqualTexts:
                # Increase w untill there are no names that start the same
                allNames = self._getAllNames()
                hasSimilarNames = True
                diff = 2
                w -= 1
                while hasSimilarNames and w < len(name):
                    w += 1
                    w2 = w - (diff - 1)
                    shortName = name[:w2]
                    similarnames = [n for n in allNames if n[:w2] == shortName]
                    hasSimilarNames = len(similarnames) > 1
            # Check again, with corrected w
            if (w + 1) < len(name):
                name = name[:w] + ELLIPSIS
                itemReduced = True
            # Set text now
            QtWidgets.QTabBar.setTabText(self, i, name)
        # Done
        return itemReduced
class CompactTabWidget(QtWidgets.QTabWidget):
    """CompactTabWidget(parent, *args, **kwargs)
    Implements a tab widget with a tabbar that is in document mode
    and has more compact tabs that conventional tab widgets, so more
    items fit on the same space.
    Further much care is taken to ellide the names in a smart way:
      * All items are allowed the same amount of characters instead of
        that the same amount of characters is removed from all names.
      * If there are two item with the same beginning, it is made
        sure that enough characters are shown such that the names
        can be distinguished.
    The kwargs are passed to the tab bar constructor. There are a few
    keywords arguments to influence the appearance of the tabs. See the
    CompactTabBar class.
    """
    def __init__(self, *args, **kwargs):
        QtWidgets.QTabWidget.__init__(self, *args)
        # Set tab bar
        self.setTabBar(CompactTabBar(self, **kwargs))
        # Draw tabs at the top by default
        self.setTabPosition(QtWidgets.QTabWidget.North)
    def setTabData(self, i, data):
        """setTabData(i, data)
        Set the given object at the tab with index 1.
        """
        self.tabBar().setTabData(i, data)
    def tabData(self, i):
        """tabData(i)
        Get the tab data at item i. Always returns a Python object.
        """
        return self.tabBar().tabData(i)
    def setTabText(self, i, text):
        """setTabText(i, text)
        Set the text for tab i.
        """
        self.tabBar().setTabText(i, text)
    def tabText(self, i):
        """tabText(i)
        Get the title of the tab at index i.
        """
        return self.tabBar().tabText(i)
if __name__ == "__main__":
    w = CompactTabWidget()
    w.show()
    w.addTab(QtWidgets.QWidget(w), "aapenootjedopje")
    w.addTab(QtWidgets.QWidget(w), "aapenootjedropje")
    w.addTab(QtWidgets.QWidget(w), "noot en mies")
    w.addTab(QtWidgets.QWidget(w), "boom bijv een iep")
    w.addTab(QtWidgets.QWidget(w), "roosemarijnus")
    w.addTab(QtWidgets.QWidget(w), "vis")
    w.addTab(QtWidgets.QWidget(w), "vuurvuurvuur")
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module editor
Defines the PyzoEditor class which is used to edit documents.
This module/class also implements all the relatively low level
file loading/saving /reloading stuff.
"""
import os, sys
import re, codecs
from pyzo.qt import QtCore, QtGui, QtWidgets
qt = QtGui
from pyzo.codeeditor import Manager
from pyzo.core.menu import EditorContextMenu
from pyzo.core.baseTextCtrl import BaseTextCtrl, normalizePath
from pyzo.core.pyzoLogging import print  # noqa
import pyzo
# Set default line ending (if not set)
if not pyzo.config.settings.defaultLineEndings:
    if sys.platform.startswith("win"):
        pyzo.config.settings.defaultLineEndings = "CRLF"
    else:
        pyzo.config.settings.defaultLineEndings = "LF"
def determineEncoding(bb):
    """Get the encoding used to encode a file.
    Accepts the bytes of the file. Returns the codec name. If the
    codec could not be determined, uses UTF-8.
    """
    # Init
    firstTwoLines = bb.split(b"\n", 2)[:2]
    encoding = "UTF-8"
    for line in firstTwoLines:
        # Try to make line a string
        try:
            line = line.decode("ASCII").strip()
        except Exception:
            continue
        # Has comment?
        if line and line[0] == "#":
            # Matches regular expression given in PEP 0263?
            expression = r"coding[:=]\s*([-\w.]+)"
            result = re.search(expression, line)
            if result:
                # Is it a known encoding? Correct name if it is
                candidate_encoding = result.group(1)
                try:
                    c = codecs.lookup(candidate_encoding)
                    candidate_encoding = c.name
                except Exception:
                    pass
                else:
                    encoding = candidate_encoding
    # Done
    return encoding
def determineLineEnding(text):
    """Get the line ending style used in the text.
    \n, \r, \r\n,
    The EOLmode is determined by counting the occurrences of each
    line ending...
    """
    text = text[:32768]  # Limit search for large files
    # test line ending by counting the occurrence of each
    c_win = text.count("\r\n")
    c_mac = text.count("\r") - c_win
    c_lin = text.count("\n") - c_win
    # set the appropriate style
    if c_win > c_mac and c_win > c_lin:
        mode = "\r\n"
    elif c_mac > c_win and c_mac > c_lin:
        mode = "\r"
    else:
        mode = "\n"
    # return
    return mode
def determineIndentation(text):
    return determineIndentationAndTrailingWS(text)[0]
def determineIndentationAndTrailingWS(text):
    """Get the indentation used in this document and whether there is
    any trailing whitespace.
    The text is analyzed to find the most used indentations.
    The result is -1 if tab indents are most common.
    A positive result means spaces are used; the amount
    signifies the amount of spaces per indentation.
    0 is returned if the indentation could not be determined.
    The second return value is the number of lines with trailing ws.
    """
    text = text[:32768]  # Limit search for large files
    # create dictionary of indents, -1 means a tab
    indents = {}
    indents[-1] = 0
    trailing = 0
    lines = text.splitlines()
    lines.insert(0, "")  # so the lines start at 1
    for i in range(len(lines)):
        line = lines[i]
        # remove indentation
        lineA = line.lstrip()
        lineB = line.rstrip()
        lineC = lineA.rstrip()
        indent = len(line) - len(lineA)
        if len(lineB) < len(line):
            trailing += 1
        line = lineC
        if line.startswith("#"):
            continue
        else:
            # remove everything after the #
            line = line.split("#", 1)[0].rstrip()
        if not line:
            # continue of no line left
            continue
        # a colon means there will be an indent
        # check the next line (or the one thereafter)
        # and calculate the indentation difference with THIS line.
        if line.endswith(":"):
            if len(lines) > i + 2:
                line2 = lines[i + 1]
                tmp = line2.lstrip()
                if not tmp:
                    line2 = lines[i + 2]
                    tmp = line2.lstrip()
                if tmp:
                    ind2 = len(line2) - len(tmp)
                    ind3 = ind2 - indent
                    if line2.startswith("\t"):
                        indents[-1] += 1
                    elif ind3 > 0:
                        if ind3 not in indents:
                            indents[ind3] = 1
                        indents[ind3] += 1
    # find which was the most common tab width.
    indent, maxvotes = 0, 0
    for nspaces in indents:
        if indents[nspaces] > maxvotes:
            indent, maxvotes = nspaces, indents[nspaces]
    # print "found tabwidth %i" % indent
    return indent, trailing
# To give each new file a unique name
newFileCounter = 0
def createEditor(parent, filename=None):
    """Tries to load the file given by the filename and
    if succesful, creates an editor instance to put it in,
    which is returned.
    If filename is None, an new/unsaved/temp file is created.
    """
    if filename is None:
        # Increase counter
        global newFileCounter
        newFileCounter += 1
        # Create editor
        editor = PyzoEditor(parent)
        editor.document().setModified(True)
        editor.removeTrailingWS = True
        # Set name
        editor._name = "<tmp {}>".format(newFileCounter)
    else:
        # check and normalize
        if not os.path.isfile(filename):
            raise IOError("File does not exist '%s'." % filename)
        # load file (as bytes)
        with open(filename, "rb") as f:
            bb = f.read()
            f.close()
        # convert to text, be gentle with files not encoded with utf-8
        encoding = determineEncoding(bb)
        text = bb.decode(encoding, "replace")
        # process line endings
        lineEndings = determineLineEnding(text)
        # if we got here safely ...
        # create editor and set text
        editor = PyzoEditor(parent)
        editor.setPlainText(text)
        editor.lineEndings = lineEndings
        editor.encoding = encoding
        editor.document().setModified(False)
        # store name and filename
        editor._filename = filename
        editor._name = os.path.split(filename)[1]
        # process indentation and trailing
        indentWidth, trailing = determineIndentationAndTrailingWS(text)
        editor.removeTrailingWS = trailing < 10  # not too much for ugly diffs
        if indentWidth == -1:  # Tabs
            editor.setIndentWidth(pyzo.config.settings.defaultIndentWidth)
            editor.setIndentUsingSpaces(False)
        elif indentWidth:
            editor.setIndentWidth(indentWidth)
            editor.setIndentUsingSpaces(True)
    if editor._filename:
        editor._modifyTime = os.path.getmtime(editor._filename)
    # Set parser
    if editor._filename:
        ext = os.path.splitext(editor._filename)[1]
        parser = Manager.suggestParser(ext, text)
        editor.setParser(parser)
    else:
        # todo: rename style -> parser
        editor.setParser(pyzo.config.settings.defaultStyle)
    # return
    return editor
class PyzoEditor(BaseTextCtrl):
    # called when dirty changed or filename changed, etc
    somethingChanged = QtCore.Signal()
    def __init__(self, parent, **kwds):
        super().__init__(parent, showLineNumbers=True, **kwds)
        # Init filename and name
        self._filename = ""
        self._name = "<TMP>"
        # View settings
        # TODO: self.setViewWrapSymbols(view.showWrapSymbols)
        self.setShowLineEndings(pyzo.config.view.showLineEndings)
        self.setShowIndentationGuides(pyzo.config.view.showIndentationGuides)
        #
        self.setWrap(bool(pyzo.config.view.wrap))
        self.setHighlightCurrentLine(pyzo.config.view.highlightCurrentLine)
        self.setLongLineIndicatorPosition(pyzo.config.view.edgeColumn)
        # TODO: self.setFolding( int(view.codeFolding)*5 )
        # bracematch is set in baseTextCtrl, since it also applies to shells
        # dito for zoom and tabWidth
        # Set line endings to default
        self.lineEndings = pyzo.config.settings.defaultLineEndings
        # Set encoding to default
        self.encoding = "UTF-8"
        # Modification time to test file change
        self._modifyTime = 0
        self.modificationChanged.connect(self._onModificationChanged)
        # To see whether the doc has changed to update the parser.
        self.textChanged.connect(self._onModified)
        # This timer is used to hide the marker that shows which code is
executed
        self._showRunCursorTimer = QtCore.QTimer()
        # Add context menu (the offset is to prevent accidental auto-clicking)
        self._menu = EditorContextMenu(self)
        self.setContextMenuPolicy(QtCore.Qt.CustomContextMenu)
        self.customContextMenuRequested.connect(
            lambda p: self._menu.popup(self.mapToGlobal(p) + QtCore.QPoint(0,
3))
        )
        # Update status bar
        self.cursorPositionChanged.connect(self._updateStatusBar)
    ## Properties
    @property
    def name(self):
        return self._name
    @property
    def filename(self):
        return self._filename
    @property
    def lineEndings(self):
        """
        Line-endings style of this file. Setter accepts machine-readable (e.g.
'\r') and human-readable (e.g. 'CR') input
        """
        return self._lineEndings
    @lineEndings.setter
    def lineEndings(self, value):
        if value in ("\r", "\n", "\r\n"):
            self._lineEndings = value
            return
        try:
            self._lineEndings = {"CR": "\r", "LF": "\n", "CRLF": "\r\n"}[value]
        except KeyError:
            raise ValueError("Invalid line endings style %r" % value)
    @property
    def lineEndingsHumanReadable(self):
        """
        Current line-endings style, human readable (e.g. 'CR')
        """
        return {"\r": "CR", "\n": "LF", "\r\n": "CRLF"}[self.lineEndings]
    @property
    def encoding(self):
        """Encoding used to convert the text of this file to bytes."""
        return self._encoding
    @encoding.setter
    def encoding(self, value):
        # Test given value, correct name if it exists
        try:
            c = codecs.lookup(value)
            value = c.name
        except Exception:
            value = codecs.lookup("UTF-8").name
        # Store
        self._encoding = value
    ##
    def justifyText(self):
        """Overloaded version of justifyText to make it use our
        configurable justificationwidth.
        """
        super().justifyText(pyzo.config.settings.justificationWidth)
    def showRunCursor(self, cursor):
        """
        Momentarily highlight a piece of code to show that this is being
executed
        """
        extraSelection = QtWidgets.QTextEdit.ExtraSelection()
        extraSelection.cursor = cursor
        extraSelection.format.setBackground(QtCore.Qt.gray)
        self.setExtraSelections([extraSelection])
        self._showRunCursorTimer.singleShot(200, lambda:
self.setExtraSelections([]))
```

```
    def id(self):
        """Get an id of this editor. This is the filename,
        or for tmp files, the name."""
        if self._filename:
            return self._filename
        else:
            return self._name
    def focusInEvent(self, event):
        """Test whether the file has been changed 'behind our back'"""
        self.testWhetherFileWasChanged()
        return super().focusInEvent(event)
    def testWhetherFileWasChanged(self):
        """testWhetherFileWasChanged()
        Test to see whether the file was changed outside our backs,
        and let the user decide what to do.
        Returns True if it was changed.
        """
        # get the path
        path = self._filename
        if not os.path.isfile(path):
            # file is deleted from the outside
            return
        # test the modification time...
        mtime = os.path.getmtime(path)
        if mtime != self._modifyTime:
            # ask user
            dlg = QtWidgets.QMessageBox(self)
            dlg.setWindowTitle("File was changed")
            dlg.setText(
                "File has been modified outside of the editor:\n" +
self._filename
            )
            dlg.setInformativeText("Do you want to reload?")
            t = dlg.addButton("Reload", QtWidgets.QMessageBox.AcceptRole)  # 0
            dlg.addButton("Keep this version", QtWidgets.QMessageBox.RejectRole)
# 1
            dlg.setDefaultButton(t)
            # whatever the result, we will reset the modified time
            self._modifyTime = os.path.getmtime(path)
            # get result and act
            result = dlg.exec_()
            if result == 0:  # in PySide6 AcceptRole != 0
                self.reload()
            else:
                pass  # when cancelled or explicitly said, do nothing
            # Return that indeed the file was changes
            return True
    def _onModificationChanged(self, changed):
        """Handler for the modificationChanged signal. Emit somethingChanged
        for the editorStack to update the modification notice."""
        self.somethingChanged.emit()
    def _onModified(self):
        pyzo.parser.parseThis(self)
    def _updateStatusBar(self):
        editor = pyzo.editors.getCurrentEditor()
        sb = pyzo.main.statusBar()
        sb.updateCursorInfo(editor)
        sb.updateFileEncodingInfo(editor)
    def dragMoveEvent(self, event):
        """Otherwise cursor can get stuck.
        https://bitbucket.org/iep-project/iep/issue/252
        https://qt-project.org/forums/viewthread/3180
        """
        if event.mimeData().hasUrls():
            event.acceptProposedAction()
        else:
            BaseTextCtrl.dropEvent(self, event)
    def dropEvent(self, event):
        """Drop files in the list."""
        if event.mimeData().hasUrls():
            # file: let the editorstack do the work.
            pyzo.editors.dropEvent(event)
        else:
            # text: act normal
            BaseTextCtrl.dropEvent(self, event)
    def showEvent(self, event=None):
        """Capture show event to change title."""
        # Act normally
        if event:
            BaseTextCtrl.showEvent(self, event)
        # Make parser update
        pyzo.parser.parseThis(self)
    def setTitleInMainWindow(self):
        """set the title  text in the main window to show filename."""
        # compose title
        name, path = self._name, self._filename
        if path:
            pyzo.main.setMainTitle(path)
        else:
            pyzo.main.setMainTitle(name)
    def save(self, filename=None):
        """Save the file. No checking is done."""
        # get filename
        if filename is None:
            filename = self._filename
        if not filename:
            raise ValueError("No filename specified, and no filename known.")
        # Test whether it was changed without us knowing. If so, dont save now.
        if self.testWhetherFileWasChanged():
            return
        # Remove whitespace in a single undo-able action
        if (
            self.removeTrailingWS
            or pyzo.config.settings.removeTrailingWhitespaceWhenSaving
        ):
            # Original cursor to put state back at the end
            oricursor = self.textCursor()
            # Screen cursor to select document
            screenCursor = self.textCursor()
            screenCursor.movePosition(screenCursor.Start)
            screenCursor.movePosition(screenCursor.End, screenCursor.KeepAnchor)
            # Cursor for doing the editor
            editCursor = self.textCursor()
            # Go!
            editCursor.beginEditBlock()
            try:
                editCursor.setPosition(screenCursor.selectionStart())
                editCursor.movePosition(editCursor.StartOfBlock)
                while (
                    editCursor.position() < screenCursor.selectionEnd()
                    or editCursor.position() <= screenCursor.selectionStart()
                ):
                    editCursor.movePosition(editCursor.StartOfBlock)
                    editCursor.movePosition(
                        editCursor.EndOfBlock, editCursor.KeepAnchor
                    )
                    text1 = editCursor.selectedText()
                    text2 = text1.rstrip()
                    if len(text1) != len(text2):
```

```
                    editCursor.insertText(text2)
                if not editCursor.block().next().isValid():
                    break
                editCursor.movePosition(editCursor.NextBlock)
        finally:
            self.setTextCursor(oricursor)
            editCursor.endEditBlock()
    # Get text and convert line endings
    text = self.toPlainText()
    text = text.replace("\n", self.lineEndings)
    # Make bytes
    bb = text.encode(self.encoding)
    # Store
    f = open(filename, "wb")
    try:
        f.write(bb)
    finally:
        f.close()
    # Update stats
    self._filename = normalizePath(filename)
    self._name = os.path.split(self._filename)[1]
    self.document().setModified(False)
    self._modifyTime = os.path.getmtime(self._filename)
    # update title (in case of a rename)
    self.setTitleInMainWindow()
    # allow item to update its texts (no need: onModifiedChanged does this)
    # self.somethingChanged.emit()
def reload(self):
    """Reload text using the self._filename.
    We do not have a load method; we first try to load the file
    and only when we succeed create an editor to show it in...
    This method is only for reloading in case the file was changed
    outside of the editor."""
    # We can only load if the filename is known
    if not self._filename:
        return
    filename = self._filename
    # Remember where we are
    cursor = self.textCursor()
    linenr = cursor.blockNumber() + 1
    # Load file (as bytes)
    with open(filename, "rb") as f:
        bb = f.read()
    # Convert to text
    text = bb.decode("UTF-8")
    # Process line endings (before setting the text)
    self.lineEndings = determineLineEnding(text)
    # Set text
    self.setPlainText(text)
    self.document().setModified(False)
    # Go where we were (approximately)
    self.gotoLine(linenr)
def deleteLines(self):
    cursor = self.textCursor()
    # Find start and end of selection
    start = cursor.selectionStart()
    end = cursor.selectionEnd()
    # Expand selection: from start of first block to start of next block
    cursor.setPosition(start)
    cursor.movePosition(cursor.StartOfBlock)
    cursor.setPosition(end, cursor.KeepAnchor)
    cursor.movePosition(cursor.NextBlock, cursor.KeepAnchor)
    cursor.removeSelectedText()
def duplicateLines(self):
    cursor = self.textCursor()
    # Find start and end of selection
    start = cursor.selectionStart()
    end = cursor.selectionEnd()
    # Expand selection: from start of first block to start of next block
    cursor.setPosition(start)
    cursor.movePosition(cursor.StartOfBlock)
    cursor.setPosition(end, cursor.KeepAnchor)
    cursor.movePosition(cursor.NextBlock, cursor.KeepAnchor)
    text = cursor.selectedText()
    cursor.setPosition(start)
    cursor.movePosition(cursor.StartOfBlock)
    cursor.insertText(text)
def commentCode(self):
    """
    Comment the lines that are currently selected
    """
    indents = []
    indentChar = " " if self.indentUsingSpaces() else "\t"
    def getIndent(cursor):
        text = cursor.block().text().rstrip()
        if text:
            indents.append(len(text) - len(text.lstrip()))
    def commentBlock(cursor):
        blockText = cursor.block().text()
        numMissingIndentChars = minindent - (
            len(blockText) - len(blockText.lstrip(indentChar))
        )
        if numMissingIndentChars > 0:
            # Prevent setPosition from leaving bounds of the current block
            # if there are too few indent characters (e.g. an empty line)
            cursor.insertText(indentChar * numMissingIndentChars)
        cursor.setPosition(cursor.block().position() + minindent)
        cursor.insertText("# ")
    self.doForSelectedBlocks(getIndent)
    minindent = min(indents) if indents else 0
    self.doForSelectedBlocks(commentBlock)
def uncommentCode(self):
    """
    Uncomment the lines that are currently selected
    """
    # TODO: this should not be applied to lines that are part of a multi-
line string
    # Define the uncomment function to be applied to all blocks
    def uncommentBlock(cursor):
        """
        Find the first # on the line; if there is just whitespace before it,
        remove the # and if it is followed by a space remove the space, too
        """
        text = cursor.block().text()
        commentStart = text.find("#")
        if commentStart == -1:
            return  # No comment on this line
        if text[:commentStart].strip() != "":
            return  # Text before the #
        # Move the cursor to the beginning of the comment
        cursor.setPosition(cursor.block().position() + commentStart)
        cursor.deleteChar()
        if text[commentStart:].startswith("# "):
            cursor.deleteChar()
    # Apply this function to all blocks
    self.doForSelectedBlocks(uncommentBlock)
def toggleCommentCode(self):
    def toggleComment():
        """
        Toggles comments for the seclected text in editor, most of the code
is
        taken from commentCode and uncommentCode
        """
        text_block = []
        def getBlocks(cursor):
            text = cursor.block().text()
            text_block.append(text)
        def commentBlock(cursor):
            cursor.setPosition(cursor.block().position())
            cursor.insertText("# ")
        def uncommentBlock(cursor):
            """
            Find the first # on the line; if there is just whitespace before
it,
            remove the # and if it is followed by a space remove the space,
too
            """
            cursor.setPosition(cursor.block().position())
            cursor.deleteChar()
            cursor.deleteChar()
        self.doForSelectedBlocks(getBlocks)
        commented = [item for item in text_block if item.startswith("# ")]
        if len(commented) == len(text_block):
            self.doForSelectedBlocks(uncommentBlock)
        else:
            self.doForSelectedBlocks(commentBlock)
    toggleComment()
def gotoDef(self):
    """
    Goto the definition for the word under the cursor
    """
    # Get name of object to go to
    cursor = self.textCursor()
    if not cursor.hasSelection():
        cursor.select(cursor.WordUnderCursor)
    word = cursor.selection().toPlainText()
    # Send the open command to the shell
    s = pyzo.shells.getCurrentShell()
    if s is not None:
        if word and word.isidentifier():
            s.executeCommand("open %s\n" % word)
        else:
            s.write("Invalid identifier %r\n" % word)
## Introspection processing methods
def processCallTip(self, cto):
    """Processes a calltip request using a CallTipObject instance."""
    # Try using buffer first
    if cto.tryUsingBuffer():
        return
    # Try obtaining calltip from the source
    sig = pyzo.parser.getFictiveSignature(cto.name, self, True)
    if sig:
        # Done
        cto.finish(sig)
    else:
        # Try the shell
        shell = pyzo.shells.getCurrentShell()
        if shell:
            shell.processCallTip(cto)
def processAutoComp(self, aco):
    """Processes an autocomp request using an AutoCompObject instance."""
    # Try using buffer first
    if aco.tryUsingBuffer():
        return
    # Init name to poll by remote process (can be changed!)
    nameForShell = aco.name
    # Get normal fictive namespace
    fictiveNS = pyzo.parser.getFictiveNameSpace(self)
    fictiveNS = set(fictiveNS)
    # Add names
    if not aco.name:
        # "root" names
        aco.addNames(fictiveNS)
        # imports
        importNames, importLines = pyzo.parser.getFictiveImports(self)
        aco.addNames(importNames)
    else:
        # Prepare list of class names to check out
        classNames = [aco.name]
        handleSelf = True
        # Unroll supers
        while classNames:
            className = classNames.pop(0)
            if not className:
                continue
            if handleSelf or (className in fictiveNS):
                # Only the self list (only first iter)
                fictiveClass = pyzo.parser.getFictiveClass(
                    className, self, handleSelf
                )
                handleSelf = False
                if fictiveClass:
                    aco.addNames(fictiveClass.members)
                    classNames.extend(fictiveClass.supers)
            else:
                nameForShell = className
                break
    # If there's a shell, let it finish the autocompletion
    shell = pyzo.shells.getCurrentShell()
    if shell:
        aco.name = nameForShell  # might be the same or a base class
        shell.processAutoComp(aco)
    else:
        # Otherwise we finish it ourselves
        aco.finish()
if __name__ == "__main__":
# Do some stubbing to run this module as a unit separate from pyzo
# TODO: untangle pyzo from this module where possible
class DummyParser:
    def parseThis(self, x):
        pass
pyzo.parser = DummyParser()
EditorContextMenu = QtWidgets.QMenu  # noqa
app = QtWidgets.QApplication([])
win = PyzoEditor(None)
QtWidgets.QShortcut(QtGui.QKeySequence("Ctrl+C"),
win).activated.connect(win.copy)
QtWidgets.QShortcut(QtGui.QKeySequence("Ctrl+X"),
win).activated.connect(win.cut)
QtWidgets.QShortcut(QtGui.QKeySequence("Ctrl+V"),
win).activated.connect(win.paste)
QtWidgets.QShortcut(QtGui.QKeySequence("Ctrl+Shift+V"),
win).activated.connect(
    win.pasteAndSelect
)
QtWidgets.QShortcut(QtGui.QKeySequence("Ctrl+Z"),
win).activated.connect(win.undo)
QtWidgets.QShortcut(QtGui.QKeySequence("Ctrl+Y"),
win).activated.connect(win.redo)
tmp = "foo(bar)\nfor bar in range(5):\n  print bar\n"
tmp += "\nclass aap:\n  def monkey(self):\n    pass\n\n"
win.setPlainText(tmp)
win.show()
app.exec_()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" EditorTabs class
Replaces the earlier EditorStack class.
The editor tabs class represents the different open files. They can
be selected using a tab widget (with tabs placed north of the editor).
It also has a find/replace widget that is at the bottom of the editor.
"""
import os, time, gc
from pyzo.qt import QtCore, QtGui, QtWidgets
import pyzo
from pyzo.core.compactTabWidget import CompactTabWidget
from pyzo.core.editor import createEditor
from pyzo.core.baseTextCtrl import normalizePath
from pyzo.core.pyzoLogging import print
from pyzo.core.icons import EditorTabToolButton
from pyzo import translate
# Constants for the alignments of tabs
MIN_NAME_WIDTH = 50
MAX_NAME_WIDTH = 200
def simpleDialog(item, action, question, options, defaultOption):
    """simpleDialog(editor, action, question, options, defaultOption)
    Options with special buttons
    ----------------------------
    ok, open, save, cancel, close, discard, apply, reset, restoredefaults,
    help, saveall, yes, yestoall, no, notoall, abort, retry, ignore.
    Returns the selected option as a string, or None if canceled.
    """
    # Get filename
    if isinstance(item, FileItem):
        filename = item.id
    else:
        filename = item.id()
    # create button map
    mb = QtWidgets.QMessageBox
    M = {
        "ok": mb.Ok,
        "open": mb.Open,
        "save": mb.Save,
        "cancel": mb.Cancel,
        "close": mb.Close,
        "discard": mb.Discard,
        "apply": mb.Apply,
        "reset": mb.Reset,
        "restoredefaults": mb.RestoreDefaults,
        "help": mb.Help,
        "saveall": mb.SaveAll,
        "yes": mb.Yes,
        "yestoall": mb.YesToAll,
        "no": mb.No,
        "notoall": mb.NoToAll,
        "abort": mb.Abort,
        "retry": mb.Retry,
        "ignore": mb.Ignore,
    }
    # setup dialog
    dlg = QtWidgets.QMessageBox(pyzo.main)
    dlg.setWindowTitle("Pyzo")
    dlg.setText(action + " file:\n{}".format(filename))
    dlg.setInformativeText(question)
    # process options
    buttons = {}
    for option in options:
        option_lower = option.lower()
        # Use standard button?
        if option_lower in M:
            button = dlg.addButton(M[option_lower])
        else:
            button = dlg.addButton(option, dlg.AcceptRole)
        buttons[button] = option
        # Set as default?
        if option_lower == defaultOption.lower():
            dlg.setDefaultButton(button)
    # get result
    dlg.exec_()
    button = dlg.clickedButton()
    if button in buttons:
        return buttons[button]
    else:
        return None
def get_shortest_unique_filename(filename, filenames):
    """Get a representation of filename in a way that makes it look
    unique compared to the other given filenames. The most unique part
    of the path is used, and every directory in between that part and the
    actual filename is represented with a slash.
    """
    # Normalize and avoid having filename itself in filenames
    filename1 = filename.replace("\\", "/")
    filenames = [fn.replace("\\", "/") for fn in filenames]
    filenames = [fn for fn in filenames if fn != filename1]
    # Prepare for finding uniqueness
    nameparts1 = filename1.split("/")
    uniqueness = [len(filenames) for i in nameparts1]
    # Establish what parts of the filename are not unique when compared to
    # each entry in filenames.
    for filename2 in filenames:
        nameparts2 = filename2.split("/")
        nonunique_for_this_filename = set()
        for i in range(len(nameparts1)):
            if i < len(nameparts2):
                if nameparts2[i] == nameparts1[i]:
                    nonunique_for_this_filename.add(i)
                if nameparts2[-1 - i] == nameparts1[-1 - i]:
                    nonunique_for_this_filename.add(-i - 1)
        for i in nonunique_for_this_filename:
            uniqueness[i] -= 1
    # How unique is the filename? If its not unique at all, use only base name
    max_uniqueness = max(uniqueness[:-1])
    if max_uniqueness == 0:
        return nameparts1[-1]
    # Produce display name based on base name and last most-unique part
    displayname = nameparts1[-1]
    for i in reversed(range(len(uniqueness) - 1)):
        displayname = "/" + displayname
        if uniqueness[i] == max_uniqueness:
            displayname = nameparts1[i] + displayname
            break
    return displayname
# todo: some management stuff could (should?) go here
class FileItem:
    """FileItem(editor)
    A file item represents an open file. It is associated with an editing
    component and has a filename.
    """
    def __init__(self, editor):
        # Store editor
        self._editor = editor
        # Init pinned state
        self._pinned = False
    @property
    def editor(self):
        """Get the editor component corresponding to this item."""
        return self._editor
    @property
    def id(self):
        """Get an id of this editor. This is the filename,
        or for tmp files, the name."""
        if self.filename:
            return self.filename
        else:
            return self.name
    @property
    def filename(self):
        """Get the full filename corresponding to this item."""
        return self._editor.filename
    @property
    def name(self):
        """Get the name corresponding to this item."""
        return self._editor.name
    @property
    def dirty(self):
        """Get whether the file has been changed since it is changed."""
        return self._editor.document().isModified()
    @property
    def pinned(self):
        """Get whether this item is pinned (i.e. will not be closed
        when closing all files.
        """
        return self._pinned
# todo: when this works with the new editor, put in own module.
class FindReplaceWidget(QtWidgets.QFrame):
    """A widget to find and replace text."""
    def __init__(self, *args):
        QtWidgets.QFrame.__init__(self, *args)
        self.setFocusPolicy(QtCore.Qt.ClickFocus)
        # init layout
        layout = QtWidgets.QHBoxLayout(self)
        layout.setSpacing(0)
        self.setLayout(layout)
        # Create some widgets first to realize a correct tab order
        self._hidebut = QtWidgets.QToolButton(self)
        self._findText = QtWidgets.QLineEdit(self)
        self._replaceText = QtWidgets.QLineEdit(self)
        if True:
            # Create sub layouts
            vsubLayout = QtWidgets.QVBoxLayout()
            vsubLayout.setSpacing(0)
            layout.addLayout(vsubLayout, 0)
            # Add button
            self._hidebut.setFont(QtGui.QFont("helvetica", 7))
            self._hidebut.setToolTip(translate("search", "Hide search widget
(Escape)"))
            self._hidebut.setIcon(pyzo.icons.cancel)
            self._hidebut.setIconSize(QtCore.QSize(16, 16))
            vsubLayout.addWidget(self._hidebut, 0)
            vsubLayout.addStretch(1)
        layout.addSpacing(10)
        if True:
            # Create sub layouts
            vsubLayout = QtWidgets.QVBoxLayout()
            hsubLayout = QtWidgets.QHBoxLayout()
            vsubLayout.setSpacing(0)
            hsubLayout.setSpacing(0)
            layout.addLayout(vsubLayout, 0)
            # Add find text
            self._findText.setToolTip(translate("search", "Find pattern"))
            vsubLayout.addWidget(self._findText, 0)
            vsubLayout.addLayout(hsubLayout)
            # Add previous button
            self._findPrev = QtWidgets.QToolButton(self)
            t = translate(
                "search", "Previous ::: Find previous occurrence of the
pattern."
            )
            self._findPrev.setText(t)
            self._findPrev.setToolTip(t.tt)
            hsubLayout.addWidget(self._findPrev, 0)
            hsubLayout.addStretch(1)
            # Add next button
            self._findNext = QtWidgets.QToolButton(self)
            t = translate("search", "Next ::: Find next occurrence of the
pattern.")
```

```
            self._findNext.setText(t)
            self._findNext.setToolTip(t.tt)
            # self._findNext.setDefault(True) # Not possible with tool buttons
            hsubLayout.addWidget(self._findNext, 0)
        layout.addSpacing(10)
        if True:
            # Create sub layouts
            vsubLayout = QtWidgets.QVBoxLayout()
            hsubLayout = QtWidgets.QHBoxLayout()
            vsubLayout.setSpacing(0)
            hsubLayout.setSpacing(0)
            layout.addLayout(vsubLayout, 0)
            # Add replace text
            self._replaceText.setToolTip(translate("search", "Replace pattern"))
            vsubLayout.addWidget(self._replaceText, 0)
            vsubLayout.addLayout(hsubLayout)
            # Add replace button
            t = translate("search", "Replace ::: Replace this match.")
            self._replaceBut = QtWidgets.QToolButton(self)
            self._replaceBut.setText(t)
            self._replaceBut.setToolTip(t.tt)
            hsubLayout.addWidget(self._replaceBut, 0)
            hsubLayout.addStretch(1)
            # Add replace kind combo
            self._replaceKind = QtWidgets.QComboBox(self)
            self._replaceKind.addItem(translate("search", "one"))
            self._replaceKind.addItem(translate("search", "all in this file"))
            self._replaceKind.addItem(translate("search", "all in all files"))
            hsubLayout.addWidget(self._replaceKind, 0)
        layout.addSpacing(10)
        if True:
            # Create sub layouts
            vsubLayout = QtWidgets.QVBoxLayout()
            vsubLayout.setSpacing(0)
            layout.addLayout(vsubLayout, 0)
            # Add match-case checkbox
            t = translate("search", "Match case ::: Find words that match
case.")
            self._caseCheck = QtWidgets.QCheckBox(t, self)
            self._caseCheck.setToolTip(t.tt)
            vsubLayout.addWidget(self._caseCheck, 0)
            # Add regexp checkbox
            t = translate("search", "RegExp ::: Find using regular
expressions.")
            self._regExp = QtWidgets.QCheckBox(t, self)
            self._regExp.setToolTip(t.tt)
            vsubLayout.addWidget(self._regExp, 0)
        if True:
            # Create sub layouts
            vsubLayout = QtWidgets.QVBoxLayout()
            vsubLayout.setSpacing(0)
            layout.addLayout(vsubLayout, 0)
            # Add whole-word checkbox
            t = translate("search", "Whole words ::: Find only whole words.")
            self._wholeWord = QtWidgets.QCheckBox(t, self)
            self._wholeWord.setToolTip(t.tt)
            self._wholeWord.resize(60, 16)
            vsubLayout.addWidget(self._wholeWord, 0)
            # Add autohide dropbox
            t = translate(
                "search", "Auto hide ::: Hide search/replace when unused for 10
s."
            )
            self._autoHide = QtWidgets.QCheckBox(t, self)
            self._autoHide.setToolTip(t.tt)
            self._autoHide.resize(60, 16)
            vsubLayout.addWidget(self._autoHide, 0)
        layout.addStretch(1)
        # Set placeholder texts
        for lineEdit in [self._findText, self._replaceText]:
            if hasattr(lineEdit, "setPlaceholderText"):
                lineEdit.setPlaceholderText(lineEdit.toolTip())
            lineEdit.textChanged.connect(self.autoHideTimerReset)
        # Set focus policy
        for but in [
            self._findPrev,
            self._findNext,
            self._replaceBut,
            self._caseCheck,
            self._wholeWord,
            self._regExp,
        ]:
            # but.setFocusPolicy(QtCore.Qt.ClickFocus)
            but.clicked.connect(self.autoHideTimerReset)
        # create timer objects
        self._timerBeginEnd = QtCore.QTimer(self)
        self._timerBeginEnd.setSingleShot(True)
        self._timerBeginEnd.timeout.connect(self.resetAppearance)
        #
        self._timerAutoHide = QtCore.QTimer(self)
        self._timerAutoHide.setSingleShot(False)
        self._timerAutoHide.setInterval(500)  # ms
        self._timerAutoHide.timeout.connect(self.autoHideTimerCallback)
        self._timerAutoHide_t0 = time.time()
        self._timerAutoHide.start()
        # create callbacks
        self._findText.returnPressed.connect(self.findNext)
        self._hidebut.clicked.connect(self.hideMe)
        self._findNext.clicked.connect(self.findNext)
        self._findPrev.clicked.connect(self.findPrevious)
        self._replaceBut.clicked.connect(self.replace)
        #
        self._regExp.stateChanged.connect(self.handleReplacePossible)
        # init case and regexp
        self._caseCheck.setChecked(bool(pyzo.config.state.find_matchCase))
        self._regExp.setChecked(bool(pyzo.config.state.find_regExp))
        self._wholeWord.setChecked(bool(pyzo.config.state.find_wholeWord))
        self._autoHide.setChecked(bool(pyzo.config.state.find_autoHide))
        # show or hide?
        if bool(pyzo.config.state.find_show):
            self.show()
        else:
            self.hide()
    def autoHideTimerReset(self):
        self._timerAutoHide_t0 = time.time()
    def autoHideTimerCallback(self):
        """Check whether we should hide the tool."""
        timeout = pyzo.config.advanced.find_autoHide_timeout
        if self._autoHide.isChecked():
            if (time.time() - self._timerAutoHide_t0) > timeout:  # seconds
                # Hide if editor has focus
                self._replaceKind.setCurrentIndex(0)  # set replace to "one"
                es = self.parent()  # editor stack
                editor = es.getCurrentEditor()
                if editor and editor.hasFocus():
                    self.hide()
    def hideMe(self):
        """Hide the find/replace widget."""
        self.hide()
        self._replaceKind.setCurrentIndex(0)  # set replace to "one"
        es = self.parent()  # editor stack
        # es._boxLayout.activate()
        editor = es.getCurrentEditor()
        if editor:
            editor.setFocus()
    def event(self, event):
        """Handle tab key and escape key. For the tab key we need to
        overload event instead of KeyPressEvent.
        """
        if isinstance(event, QtGui.QKeyEvent):
            if event.key() in (QtCore.Qt.Key_Tab, QtCore.Qt.Key_Backtab):
                event.accept()  # focusNextPrevChild is called by Qt
                return True
            elif event.key() == QtCore.Qt.Key_Escape:
                self.hideMe()
                event.accept()
                return True
        # Otherwise ... handle in default manner
        return QtWidgets.QFrame.event(self, event)
    def handleReplacePossible(self, state):
        """Disable replacing when using regular expressions."""
        for w in [self._replaceText, self._replaceBut, self._replaceKind]:
            w.setEnabled(not state)
    def startFind(self, event=None):
        """Use this rather than show(). It will check if anything is
        selected in the current editor, and if so, will set that as the
        initial search string
        """
        # show
        self.show()
        self.autoHideTimerReset()
        # get needle
        editor = self.parent().getCurrentEditor()
        if editor:
            needle = editor.textCursor().selectedText().replace("\u2029", "\n")
            if needle:
                self._findText.setText(needle)
        # select the find-text
        self.selectFindText()
    def notifyPassBeginEnd(self):
        self.setStyleSheet("QFrame { background:#f00; }")
        self._timerBeginEnd.start(300)
    def resetAppearance(self):
        self.setStyleSheet("QFrame {}")
    def selectFindText(self):
        """Select the textcontrol for the find needle,
        and the text in it"""
        # select text
        self._findText.selectAll()
        # focus
        self._findText.setFocus()
    def findNext(self, event=None):
        self.find()
        # self._findText.setFocus()
    def findPrevious(self, event=None):
        self.find(False)
        # self._findText.setFocus()
    def findSelection(self, event=None):
        self.startFind()
        self.findNext()
    def findSelectionBw(self, event=None):
        self.startFind()
        self.findPrevious()
    def find(self, forward=True, wrapAround=True, editor=None):
        """The main find method.
        Returns True if a match was found."""
        # Reset timer
        self.autoHideTimerReset()
        # get editor
        if not editor:
            editor = self.parent().getCurrentEditor()
            if not editor:
                return
        # find flags
        flags = QtGui.QTextDocument.FindFlags()
        if self._caseCheck.isChecked():
            flags |= QtGui.QTextDocument.FindCaseSensitively
        if not forward:
            flags |= QtGui.QTextDocument.FindBackward
        # if self._wholeWord.isChecked():
        #    flags |= QtGui.QTextDocument.FindWholeWords
        # focus
        self.selectFindText()
        # get text to find
        needle = self._findText.text()
        if self._regExp.isChecked():
            # Make needle a QRegExp; speciffy case-sensitivity here since the
            # FindCaseSensitively flag is ignored when finding using a QRegExp
            needle = QtCore.QRegExp(
                needle,
                QtCore.Qt.CaseSensitive
                if self._caseCheck.isChecked()
                else QtCore.Qt.CaseInsensitive,
            )
        elif self._wholeWord.isChecked():
            # Use regexp, because the default begaviour does not find
            # whole words correctly, see issue #276
            # it should *not* find this in this_word
            needle = QtCore.QRegExp(
                r"\b" + needle + r"\b",
                QtCore.Qt.CaseSensitive
                if self._caseCheck.isChecked()
                else QtCore.Qt.CaseInsensitive,
            )
        # estblish start position
        cursor = editor.textCursor()
        result = editor.document().find(needle, cursor, flags)
        if not result.isNull():
            editor.setTextCursor(result)
        elif wrapAround:
            self.notifyPassBeginEnd()
            # Move cursor to start or end of document
            if forward:
                cursor.movePosition(cursor.Start)
            else:
                cursor.movePosition(cursor.End)
            # Try again
            result = editor.document().find(needle, cursor, flags)
            if not result.isNull():
                editor.setTextCursor(result)
        # done
        editor.setFocus()
        return not result.isNull()
    def replace(self, event=None):
        i = self._replaceKind.currentIndex()
        if i == 0:
            self.replaceOne(event)
        elif i == 1:
            self.replaceAll(event)
        elif i == 2:
            self.replaceInAllFiles(event)
        else:
            raise RuntimeError("Unexpected kind of replace %s" % i)
    def replaceOne(self, event=None, wrapAround=True, editor=None):
        """If the currently selected text matches the find string,
        replaces that text. Then it finds and selects the next match.
        Returns True if a next match was found.
        """
        # get editor
        if not editor:
            editor = self.parent().getCurrentEditor()
            if not editor:
                return
        # Create a cursor to do the editing
        cursor = editor.textCursor()
        # matchCase
        matchCase = self._caseCheck.isChecked()
        # get text to find
        needle = self._findText.text()
        if not matchCase:
            needle = needle.lower()
        # get replacement
        replacement = self._replaceText.text()
        # get original text
        original = cursor.selectedText().replace("\u2029", "\n")
        if not original:
            original = ""
        if not matchCase:
            original = original.lower()
        # replace
        # TODO: < line does not work for regexp-search!
        if original and original == needle:
            cursor.insertText(replacement)
        # next!
        return self.find(wrapAround=wrapAround, editor=editor)
    def replaceAll(self, event=None, editor=None):
        # TODO: share a cursor between all replaces, in order to
        # make this one undo/redo-step
        # get editor
        if not editor:
            editor = self.parent().getCurrentEditor()
            if not editor:
                return
        # get current position
        originalPosition = editor.textCursor()
        # Move to beginning of text and replace all
        # Make this a single undo operation
        cursor = editor.textCursor()
        cursor.beginEditBlock()
        try:
            cursor.movePosition(cursor.Start)
            editor.setTextCursor(cursor)
            while self.replaceOne(wrapAround=False, editor=editor):
                pass
        finally:
            cursor.endEditBlock()
        # reset position
        editor.setTextCursor(originalPosition)
    def replaceInAllFiles(self, event=None):
        for editor in pyzo.editors:
            self.replaceAll(event, editor)
class FileTabWidget(CompactTabWidget):
    """FileTabWidget(parent)
    The tab widget that contains the editors and lists all open files.
    """
    def __init__(self, parent):
        CompactTabWidget.__init__(self, parent, padding=(2, 1, 0, 4))
        # Init main file
        self._mainFile = ""
        # Init item history
        self._itemHistory = []
        #         # Create a corner widget
        #         but = QtWidgets.QToolButton()
        #         but.setIcon( pyzo.icons.cross )
        #         but.setIconSize(QtCore.QSize(16,16))
        #         but.clicked.connect(self.onClose)
        #         self.setCornerWidget(but)
        # Bind signal to update items and keep track of history
        self.currentChanged.connect(self.updateItems)
        self.currentChanged.connect(self.trackHistory)
        self.currentChanged.connect(self.setTitleInMainWindowWhenTabChanged)
        self.setTitleInMainWindowWhenTabChanged(-1)
    def setTitleInMainWindowWhenTabChanged(self, index):
        # Valid index?
        if index < 0 or index >= self.count():
            pyzo.main.setMainTitle()  # No open file
        # Remove current item from history
        currentItem = self.currentItem()
        if currentItem:
            currentItem.editor.setTitleInMainWindow()
    ## Item management
    def items(self):
        """Get the items in the tab widget. These are Item instances, and
        are in the order in which they are at the tab bar.
        """
        tabBar = self.tabBar()
        items = []
        for i in range(tabBar.count()):
            item = tabBar.tabData(i)
            if item is None:
                continue
            items.append(item)
        return items
    def currentItem(self):
        """Get the item corresponding to the currently active tab."""
        i = self.currentIndex()
        if i >= 0:
            return self.tabBar().tabData(i)
    def getItemAt(self, i):
        return self.tabBar().tabData(i)
    def mainItem(self):
        """Get the item corresponding to the "main" file. Returns None
        if there is no main file.
        """
        for item in self.items():
            if item.id == self._mainFile:
                return item
        else:
            return None
    def trackHistory(self, index):
        """trackHistory(index)
        Called when a tab is changed. Puts the current item on top of
        the history.
        """
        # Valid index?
        if index < 0 or index >= self.count():
            return
        # Remove current item from history
        currentItem = self.currentItem()
        while currentItem in self._itemHistory:
            self._itemHistory.remove(currentItem)
        # Add current item to history
        self._itemHistory.insert(0, currentItem)
        # Limit history size
        self._itemHistory[10:] = []
    def setCurrentItem(self, item):
        """_setCurrentItem(self, item)
        Set a FileItem instance to be the current. If the given item
        is not in the list, no action is taken.
        item can be an int, FileItem, or file name.
        """
        if isinstance(item, int):
            self.setCurrentIndex(item)
        elif isinstance(item, FileItem):
            items = self.items()
            for i in range(self.count()):
                if item is items[i]:
                    self.setCurrentIndex(i)
                    break
        elif isinstance(item, str):
            items = self.items()
            for i in range(self.count()):
                if item == items[i].filename:
                    self.setCurrentIndex(i)
                    break
        else:
            raise ValueError("item should be int, FileItem or file name.")
    def selectPreviousItem(self):
        """Select the previously selected item."""
        # make an old item history
        if len(self._itemHistory) > 1 and self._itemHistory[1] is not None:
            item = self._itemHistory[1]
            self.setCurrentItem(item)
        # just select first one then ...
        elif self.count():
            item = 0
            self.setCurrentItem(item)
    ## Closing, adding and updating
    def onClose(self):
        """onClose()
        Request to close the current tab.
        """
        self.tabCloseRequested.emit(self.currentIndex())
    def removeTab(self, which):
        """removeTab(which)
        Removes the specified tab. which can be an integer, an item,
        or an editor.
        """
        # Init
        items = self.items()
        theIndex = -1
        # Find index
        if isinstance(which, int) and which >= 0 and which < len(items):
            theIndex = which
        elif isinstance(which, FileItem):
            for i in range(self.count()):
                if items[i] is which:
                    theIndex = i
                    break
        elif isinstance(which, str):
            for i in range(self.count()):
                if items[i].filename == which:
                    theIndex = i
                    break
        elif hasattr(which, "_filename"):
            for i in range(self.count()):
                if items[i].filename == which._filename:
                    theIndex = i
                    break
        else:
            raise ValueError(
                "removeTab accepts a FileItem, integer, file name, or editor."
            )
        if theIndex >= 0:
            # Close tab
            CompactTabWidget.removeTab(self, theIndex)
            # Delete editor
            items[theIndex].editor.destroy()
            gc.collect()
    def addItem(self, item, update=True):
        """addItem(item, update=True)
        Add item to the tab widget. Set update to false if you are
        calling this method many times in a row. Then use updateItemsFull()
        to update the tab widget.
        """
        # Add tab and widget
        i = self.addTab(item.editor, item.name)
        tabBut = EditorTabToolButton(self.tabBar())
        self.tabBar().setTabButton(i, QtWidgets.QTabBar.LeftSide, tabBut)
        # Keep informed about changes
        item.editor.somethingChanged.connect(self.updateItems)
        item.editor.blockCountChanged.connect(self.updateItems)
        item.editor.breakPointsChanged.connect(self.parent().updateBreakPoints)
        # Store the item at the tab
        self.tabBar().setTabData(i, item)
        # Emit the currentChanged again (already emitted on addTab), because
        # now the itemdata is actually set
        self.currentChanged.emit(self.currentIndex())
        # Update
        if update:
            self.updateItems()
    def updateItemsFull(self):
        """updateItemsFull()
        Update the appearance of the items and also updates names and
        re-aligns the items.
        """
        self.updateItems()
        self.tabBar().alignTabs()
    def updateItems(self):
        """updateItems()
        Update the appearance of the items.
        """
        # Get items and tab bar
        items = self.items()
        tabBar = self.tabBar()
        # Check whether we have name clashes, which we can try to resolve
        namecounts = {}
        for i in range(len(items)):
            item = items[i]
            if item is None:
                continue
            xx = namecounts.setdefault(item.name, [])
            xx.append(item)
        for i in range(len(items)):
            # Get item
            item = items[i]
            if item is None:
                continue
            # Get display name
            items_with_this_name = namecounts[item.name]
            if len(items_with_this_name) <= 1:
                display_name = item.name
            else:
                filenames = [j.filename for j in items_with_this_name]
                try:
                    display_name = get_shortest_unique_filename(
                        item.filename, filenames
                    )
                except Exception as err:
                    # Catch this, just in case ...
                    print("could not get unique name for:\n%r" % filenames)
                    print(err)
                    display_name = item.name
            tabBar.setTabText(i, display_name)
            # Update name and tooltip
            if item.dirty:
                tabBar.setTabToolTip(i, item.filename + " [modified]")
            else:
                tabBar.setTabToolTip(i, item.filename)
            # Determine text color. Is main file? Is current?
            if self._mainFile == item.id:
                tabBar.setTabTextColor(i, QtGui.QColor("#008"))
            elif i == self.currentIndex():
                tabBar.setTabTextColor(i, QtGui.QColor("#000"))
            else:
                tabBar.setTabTextColor(i, QtGui.QColor("#444"))
            # Get number of blocks
            nBlocks = item.editor.blockCount()
            if nBlocks == 1 and not item.editor.toPlainText():
                nBlocks = 0
            # Update appearance of icon
            but = tabBar.tabButton(i, QtWidgets.QTabBar.LeftSide)
            but.updateIcon(item.dirty, self._mainFile == item.id, item.pinned,
nBlocks)
class EditorTabs(QtWidgets.QWidget):
    """The EditorTabs instance manages the open files and corresponding
    editors. It does the saving loading etc.
    """
    # Signal to indicate that a breakpoint has changed, emits dict
    breakPointsChanged = QtCore.Signal(object)
    # Signal to notify that a different file was selected
    currentChanged = QtCore.Signal()
    # Signal to notify that the parser has parsed the text (emit by parser)
    parserDone = QtCore.Signal()
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        # keep a booking of opened directories
        self._lastpath = ""
        # keep track of all breakpoints
        self._breakPoints = {}
        # create tab widget
        self._tabs = FileTabWidget(self)
        self._tabs.tabCloseRequested.connect(self.closeFile)
        self._tabs.currentChanged.connect(self.onCurrentChanged)
        # Double clicking a tab saves the file, clicking on the bar opens a new 
file
        self._tabs.tabBar().tabDoubleClicked.connect(self.saveFile)
        self._tabs.tabBar().barDoubleClicked.connect(self.newFile)
        # Create find/replace widget
        self._findReplace = FindReplaceWidget(self)
        # create box layout control and add widgets
        self._boxLayout = QtWidgets.QVBoxLayout(self)
        self._boxLayout.addWidget(self._tabs, 1)
        self._boxLayout.addWidget(self._findReplace, 0)
        # spacing of widgets
        self._boxLayout.setSpacing(0)
        # set margins
        margin = pyzo.config.view.widgetMargin
        self._boxLayout.setContentsMargins(margin, margin, margin, margin)
        # apply
        self.setLayout(self._boxLayout)
        # self.setAttribute(QtCore.Qt.WA_AlwaysShowToolTips,True)
        # accept drops
        self.setAcceptDrops(True)
        # restore state (call later so that the menu module can bind to the
        # currentChanged signal first, in order to set tab/indentation
        # checkmarks appropriately)
        # todo: Resetting the scrolling would work better if set after
        # the widgets are properly sized.
        pyzo.callLater(self.restoreEditorState)
    def addContextMenu(self):
        """Adds a context menu to the tab bar"""
        from pyzo.core.menu import EditorTabContextMenu
        self._menu = EditorTabContextMenu(self, "EditorTabMenu")
        self._tabs.setContextMenuPolicy(QtCore.Qt.CustomContextMenu)
        self._tabs.customContextMenuRequested.connect(self.contextMenuTriggered)
    def contextMenuTriggered(self, p):
        """Called when context menu is clicked"""
        # Get index of current tab
        index = self._tabs.tabBar().tabAt(p)
        self._menu.setIndex(index)
        # Show menu if item is available
        if index >= 0:
            p = self._tabs.tabBar().tabRect(index).bottomLeft()
            self._menu.popup(self._tabs.tabBar().mapToGlobal(p))
    def onCurrentChanged(self):
        self.currentChanged.emit()
        # Update status bar
        editor = pyzo.editors.getCurrentEditor()
        sb = pyzo.main.statusBar()
        sb.updateCursorInfo(editor)
        sb.updateFileEncodingInfo(editor)
    def getCurrentEditor(self):
        """Get the currently active editor."""
        item = self._tabs.currentItem()
        if item:
            return item.editor
        else:
            return None
    def getMainEditor(self):
        """Get the editor that represents the main file, or None if
        there is no main file."""
        item = self._tabs.mainItem()
        if item:
            return item.editor
        else:
            return None
    def __iter__(self):
        tmp = [item.editor for item in self._tabs.items()]
        return tmp.__iter__()
    def updateBreakPoints(self, editor=None):
        # Get list of editors to update keypoints for
        if editor is None:
            editors = self
            self._breakPoints = {}  # Full reset
        else:
            editors = [editor]
        # Update our keypoints dict
        for editor in editors:
            fname = editor._filename or editor._name
            if not fname:
                continue
            linenumbers = editor.breakPoints()
            if linenumbers:
                self._breakPoints[fname] = linenumbers
            else:
                self._breakPoints.pop(fname, None)
        # Emit signal so shells can update the kernel
        self.breakPointsChanged.emit(self._breakPoints)
    def setDebugLineIndicators(self, *filename_linenr):
        """Set the debug line indicator. There is one indicator
        global to pyzo, corresponding to the last shell for which we
        received the indicator.
        """
        if len(filename_linenr) and filename_linenr[0] is None:
            filename_linenr = []
        # Normalize case
        filename_linenr = [(os.path.normcase(i[0]), int(i[1])) for i in
filename_linenr]
        for item in self._tabs.items():
            # Prepare
            editor = item._editor
            fname = editor._filename or editor._name
            fname = os.path.normcase(fname)
            # Reset
            editor.setDebugLineIndicator(None)
            # Set
            for filename, linenr in filename_linenr:
                if fname == filename:
                    active = (filename, linenr) == filename_linenr[-1]
                    editor.setDebugLineIndicator(linenr, active)
    ## Loading ad saving files
    def dragEnterEvent(self, event):
        if event.mimeData().hasUrls():
            event.acceptProposedAction()
    def dropEvent(self, event):
        """Drop files in the list."""
        for qurl in event.mimeData().urls():
            path = str(qurl.toLocalFile())
            if os.path.isfile(path):
                self.loadFile(path)
            elif os.path.isdir(path):
                self.loadDir(path)
            else:
                pass
    def newFile(self):
        """Create a new (unsaved) file."""
        # create editor
        editor = createEditor(self, None)
        editor.document().setModified(False)  # Start out as OK
        # add to list
        item = FileItem(editor)
        self._tabs.addItem(item)
        self._tabs.setCurrentItem(item)
        # set focus to new file
        editor.setFocus()
        return item
    def openFile(self):
        """Create a dialog for the user to select a file."""
        # determine start dir
        # todo: better selection of dir, using project manager
        editor = self.getCurrentEditor()
        if editor and editor._filename:
            startdir = os.path.split(editor._filename)[0]
        else:
            startdir = self._lastpath
        if (not startdir) or (not os.path.isdir(startdir)):
            startdir = ""
        # show dialog
        msg = translate("editorTabs", "Select one or more files to open")
        filter = "Python (*.py *.pyw);;"
        filter += "Pyrex (*.pyi *.pyx *.pxd);;"
        filter += "C (*.c *.h *.cpp *.c++);;"
        # filter += "Py+Cy+C (*.py *.pyw *.pyi *.pyx *.pxd *.c *.h *.cpp);;"
        filter += "All (*)"
        if True:
            filenames = QtWidgets.QFileDialog.getOpenFileNames(
                self, msg, startdir, filter
            )
            if isinstance(filenames, tuple):  # PySide
                filenames = filenames[0]
        else:
            # Example how to preselect files, can be used when the users
            # opens a file in a project to select all files currently not
            # loaded.
            d = QtWidgets.QFileDialog(self, msg, startdir, filter)
            d.setFileMode(d.ExistingFiles)
            d.selectFile('"codeparser.py" "editorStack.py"')
            d.exec_()
            if d.result():
                filenames = d.selectedFiles()
            else:
                filenames = []
        # were some selected?
        if not filenames:
            return
        # load
        for filename in filenames:
            self.loadFile(filename)
    def openDir(self):
        """Create a dialog for the user to select a directory."""
        # determine start dir
        editor = self.getCurrentEditor()
        if editor and editor._filename:
            startdir = os.path.split(editor._filename)[0]
        else:
            startdir = self._lastpath
        if not os.path.isdir(startdir):
            startdir = ""
        # show dialog
        msg = "Select a directory to open"
        dirname = QtWidgets.QFileDialog.getExistingDirectory(self, msg,
startdir)
        # was a dir selected?
        if not dirname:
            return
        # load
        self.loadDir(dirname)
    def loadFile(self, filename, updateTabs=True, ignoreFail=False):
        """Load the specified file.
        On success returns the item of the file, also if it was
        already open."""
        # Note that by giving the name of a tempfile, we can select that
        # temp file.
        # normalize path
        if filename[0] != "<":
            filename = normalizePath(filename)
        if not filename:
            return None
        # if the file is already open...
        for item in self._tabs.items():
            if item.id == filename:
                # id gets _filename or _name for temp files
                break
        else:
            item = None
        if item:
            self._tabs.setCurrentItem(item)
            print("File already open: '{}'".format(filename))
            return item
        # create editor
        try:
            editor = createEditor(self, filename)
        except Exception as err:
            # Notify in logger
            print("Error loading file: ", err)
            # Make sure the user knows
            if not ignoreFail:
                m = QtWidgets.QMessageBox(self)
                m.setWindowTitle("Error loading file")
                m.setText(str(err))
                m.setIcon(m.Warning)
                m.exec_()
            return None
        # create list item
        item = FileItem(editor)
        self._tabs.addItem(item, updateTabs)
        if updateTabs:
            self._tabs.setCurrentItem(item)
        # store the path
        self._lastpath = os.path.dirname(item.filename)
        return item
    def loadDir(self, path):
        """Create a project with the dir's name and add all files
        contained in the directory to it.
        extensions is a komma separated list of extenstions of files
        to accept...
        """
        # if the path does not exist, stop
        path = os.path.abspath(path)
        if not os.path.isdir(path):
            print("ERROR loading dir: the specified directory does not exist!")
            return
        # get extensions
        extensions = pyzo.config.advanced.fileExtensionsToLoadFromDir
        extensions = extensions.replace(",", " ").replace(";", " ")
        extensions = ["." + a.lstrip(".").strip() for a in extensions.split("
")]
        # init item
        item = None
        # open all qualified files...
        self._tabs.setUpdatesEnabled(False)
        try:
            filelist = os.listdir(path)
            for filename in filelist:
                filename = os.path.join(path, filename)
                ext = os.path.splitext(filename)[1]
                if str(ext) in extensions:
                    item = self.loadFile(filename, False)
        finally:
            self._tabs.setUpdatesEnabled(True)
            self._tabs.updateItems()
        # return lastopened item
        return item
    def saveFileAs(self, editor=None):
        """Create a dialog for the user to select a file.
        returns: True if succesfull, False if fails
        """
        # get editor
        if editor is None:
            editor = self.getCurrentEditor()
        if editor is None:
            return False
        # get startdir
        if editor._filename:
            startdir = os.path.dirname(editor._filename)
        else:
            startdir = self._lastpath
            # Try the file browser or project manager to suggest a path
            fileBrowser = pyzo.toolManager.getTool("pyzofilebrowser")
            projectManager = pyzo.toolManager.getTool("pyzoprojectmanager")
            if fileBrowser:
                startdir = fileBrowser.getDefaultSavePath()
            if projectManager and not startdir:
                startdir = projectManager.getDefaultSavePath()
            if not startdir:
                startdir = os.path.expanduser("~")
        if not os.path.isdir(startdir):
            startdir = ""
        # show dialog
        msg = translate("editorTabs", "Select the file to save to")
        filter = "Python (*.py *.pyw);;"
        filter += "Pyrex (*.pyi *.pyx *.pxd);;"
        filter += "C (*.c *.h *.cpp);;"
        # filter += "Py+Cy+C (*.py *.pyw *.pyi *.pyx *.pxd *.c *.h *.cpp);;"
        filter += "All (*.*)"
        filename = QtWidgets.QFileDialog.getSaveFileName(self, msg, startdir,
filter)
        if isinstance(filename, tuple):  # PySide
            filename = filename[0]
        # give python extension if it has no extension
        head, tail = os.path.split(filename)
        if tail and "." not in tail:
            filename += ".py"
        # proceed or cancel
        if filename:
            return self.saveFile(editor, filename)
        else:
            return False  # Cancel was pressed
    def saveFile(self, editor=None, filename=None):
        """Save the file.
        returns: True if succesfull, False if fails
        """
        # get editor
        if editor is None:
            editor = self.getCurrentEditor()
        elif isinstance(editor, int):
            index = editor
            editor = None
            if index >= 0:
                item = self._tabs.items()[index]
                editor = item.editor
        if editor is None:
            return False
        # get filename
        if filename is None:
            filename = editor._filename
        if not filename:
            return self.saveFileAs(editor)
        # let the editor do the low level stuff...
        try:
            editor.save(filename)
        except Exception as err:
            # Notify in logger
            print("Error saving file:", err)
            # Make sure the user knows
            m = QtWidgets.QMessageBox(self)
            m.setWindowTitle("Error saving file")
            m.setText(str(err))
            m.setIcon(m.Warning)
            m.exec_()
            # Return now
            return False
        # get actual normalized filename
        filename = editor._filename
        # notify
        # TODO: message concerining line endings
        print("saved file: {} ({})".format(filename,
editor.lineEndingsHumanReadable))
        self._tabs.updateItems()
        # todo: this is where we once detected whether the file being saved was
a style file.
        # Notify done
        return True
    def saveAllFiles(self):
        """Save all files"""
        for editor in self:
            self.saveFile(editor)
    ## Closing files / closing down
    def _get_action_texts(self):
        options = translate("editor", "Close, Discard, Cancel, Save").split(",")
        options = [i.strip() for i in options]
        try:
            close_txt, discard_txt, cancel_txt, save_txt = options
        except Exception:
            print("error in translation for close, discard, cancel, save.")
            close_txt, discard_txt, cancel_txt, save_txt = (
                "Close",
                "Discard",
                "Cancel",
                "Save",
            )
        return close_txt, discard_txt, cancel_txt, save_txt
    def askToSaveFileIfDirty(self, editor):
        """askToSaveFileIfDirty(editor)
        If the given file is not saved, pop up a dialog
        where the user can save the file
        Returns 1 if file need not be saved.
        Returns 2 if file was saved.
        Returns 3 if user discarded changes.
        Returns 0 if cancelled.
        """
        # should we ask to save the file?
        if editor.document().isModified():
            # Ask user what to do
            close_txt, discard_txt, cancel_txt, save_txt = 
self._get_action_texts()
            result = simpleDialog(
                editor,
                translate("editor", "Closing"),
                translate("editor", "Save modified file?"),
                [discard_txt, cancel_txt, save_txt],
                save_txt,
            )
            # Get result and act
            if result == save_txt:
                return 2 if self.saveFile(editor) else 0
            elif result == discard_txt:
                return 3
            else:  # cancel
                return 0
        return 1
    def closeFile(self, editor=None):
        """Close the selected (or current) editor.
        Returns same result as askToSaveFileIfDirty()"""
        # get editor
        if editor is None:
            editor = self.getCurrentEditor()
            item = self._tabs.currentItem()
        elif isinstance(editor, int):
            index = editor
            editor, item = None, None
            if index >= 0:
                item = self._tabs.items()[index]
                editor = item.editor
        else:
            item = None
            for i in self._tabs.items():
                if i.editor is editor:
                    item = i
        if editor is None or item is None:
            return
        # Ask if dirty
        result = self.askToSaveFileIfDirty(editor)
        # Ask if closing pinned file
        close_txt, discard_txt, cancel_txt, save_txt = self._get_action_texts()
        if result and item.pinned:
            result = simpleDialog(
                editor,
                translate("editor", "Closing pinned"),
                translate("editor", "Are you sure you want to close this pinned 
file?"),
                [close_txt, cancel_txt],
                cancel_txt,
            )
            result = result == close_txt
        # ok, close...
        if result:
            if editor._name.startswith("<tmp"):
                # Temp file, try to find its index
                for i in range(len(self._tabs.items())):
                    if self._tabs.getItemAt(i).editor is editor:
                        self._tabs.removeTab(i)
                        break
            else:
                self._tabs.removeTab(editor)
        # Clear any breakpoints that it may have had
        self.updateBreakPoints()
        return result
    def closeAllFiles(self):
        """Close all files"""
        for editor in self:
            self.closeFile(editor)
    def saveEditorState(self):
        """Save the editor's state configuration."""
        fr = self._findReplace
        pyzo.config.state.find_matchCase = fr._caseCheck.isChecked()
        pyzo.config.state.find_regExp = fr._regExp.isChecked()
        pyzo.config.state.find_wholeWord = fr._wholeWord.isChecked()
        pyzo.config.state.find_show = fr.isVisible()
        #
        pyzo.config.state.editorState2 = self._getCurrentOpenFilesAsSsdfList()
    def restoreEditorState(self):
        """Restore the editor's state configuration."""
        # Restore opened editors
        if pyzo.config.state.editorState2:
            ok = 
self._setCurrentOpenFilesAsSsdfList(pyzo.config.state.editorState2)
            if not ok:
                self.newFile()
        else:
            self.newFile()
        # The find/replace state is set in the corresponding class during init
    def _getCurrentOpenFilesAsSsdfList(self):
        """Get the state as it currently is as an ssdf list.
        The state entails all open files and their structure in the
        projects. The being collapsed of projects and their main files.
        The position of the cursor in the editors.
        """
        # Init
        state = []
        # Get items
        for item in self._tabs.items():
            # Get editor
            ed = item.editor
            if not ed._filename:
                continue
            # Init info
            info = []
            # Add filename, line number, and scroll distance
            info.append(ed._filename)
            info.append(int(ed.textCursor().position()))
            info.append(int(ed.verticalScrollBar().value()))
            # Add whether pinned or main file
            if item.pinned:
                info.append("pinned")
            if item.id == self._tabs._mainFile:
                info.append("main")
            # Add to state
            state.append(tuple(info))
        # Get history
        history = [item for item in self._tabs._itemHistory]
        history.reverse()  # Last one is current
        for item in history:
            if isinstance(item, FileItem):
                ed = item._editor
                if ed._filename:
                    state.append((ed._filename, "hist"))
        # Done
        return state
    def _setCurrentOpenFilesAsSsdfList(self, state):
        """Set the state of the editor in terms of opened files.
        The input should be a list object as returned by
        ._getCurrentOpenFilesAsSsdfList().
        """
        # Init dict
        fileItems = {}
        # Process items
        for item in state:
            fname = item[0]
            if os.path.exists(fname):
                if item[1] == "hist":
                    # select item (to make the history right)
                    if fname in fileItems:
                        self._tabs.setCurrentItem(fileItems[fname])
                elif fname:
                    # a file item, create editor-item and store
                    itm = self.loadFile(fname, ignoreFail=True)
                    # set position
                    if itm:
                        fileItems[fname] = itm
                        try:
                            ed = itm.editor
                            cursor = ed.textCursor()
                            cursor.setPosition(int(item[1]))
                            ed.setTextCursor(cursor)
                            # set scrolling
                            ed.verticalScrollBar().setValue(int(item[2]))
                            # ed.centerCursor() #TODO: this does not work
properly yet
                            # set main and/or pinned?
                            if "main" in item:
                                self._tabs._mainFile = itm.id
                            if "pinned" in item:
                                itm._pinned = True
                        except Exception as err:
                            print("Could not set position for %s" % fname, err)
        return len(fileItems) != 0
    def closeAll(self):
        """Close all files (well technically, we don't really close them,
        so that they are all stil there when the user presses cancel).
        Returns False if the user pressed cancel when asked for
        saving an unsaved file.
        """
        # try closing all editors.
        for editor in self:
            result = self.askToSaveFileIfDirty(editor)
            if not result:
                return False
        # we're good to go closing
        return True
```

```
import os
import datetime
import pyzo
from pyzo.qt import QtCore, QtGui, QtWidgets  # noqa
class CommandHistory(QtCore.QObject):
    """Keep track of a (global) history of commands.
    This kinda assumes Python commands, but should be easy enough to generalize.
    """
    max_commands = 2000
    command_added = QtCore.Signal(str)
    command_removed = QtCore.Signal(int)
    commands_reset = QtCore.Signal()
    def __init__(self, fname):
        super().__init__()
        self._commands = []
        self._fname = fname
        self._last_date = datetime.date(2000, 1, 1)
        self._load()
    def _load(self):
        """Load commands from file."""
        if not self._fname:
            return
        assert not self._commands
        try:
            filename = os.path.join(pyzo.appDataDir, self._fname)
            if not os.path.isfile(filename):
                with open(filename, "wb"):
                    pass
            # Load lines and add to commands
            lines = open(filename, "r", encoding="utf-8").read().splitlines()
            self._commands.extend(
                [line.rstrip() for line in lines[-self.max_commands :]]
            )
            # Resolve last date
            for c in reversed(lines):
                if c.startswith("# ==== "):
                    try:
                        c = c.split("====")[1].strip()
                        self._last_date = datetime.datetime.strptime(
                            c, "%Y-%m-%d"
                        ).date()
                        break
                    except Exception:
```

```
                    pass
        except Exception as e:
            print("An error occurred while loading the history: " + str(e))
    def save(self):
        """Save the commands to disk."""
        if not self._fname:
            return
        filename = os.path.join(pyzo.appDataDir, self._fname)
        try:
            with open(filename, "wt", encoding="utf-8") as f:
                f.write("\n".join(self._commands))
        except Exception:
            print("Could not save command history")
    def get_commands(self):
        """Get a list of all commands (latest last)."""
        return self._commands.copy()
    def append(self, command):
        """Add a command to the list."""
        command = command.rstrip()
        if not command:
            return
        # Add date?
        today = datetime.date.today()
        if today > self._last_date:
            self._last_date = today
            self._commands.append("# ==== " + today.strftime("%Y-%m-%d"))
            self.command_added.emit(self._commands[-1])
        # Clear it
        try:
            index = self._commands.index(command)
        except ValueError:
            pass
        else:
            self._commands.pop(index)
            self.command_removed.emit(index)
        # Append
        self._commands.append(command)
        self.command_added.emit(self._commands[-1])
        # Reset?
        if len(self._commands) > self.max_commands:
            self._commands[: self.max_commands // 2] = []
            self.commands_reset.emit()
    def pop(self, index):
        """Remove a command by index."""
        self._commands.pop(index)
        self.command_removed.emit(index)
    def find_starting_with(self, firstpart, n=1):
        """Find the nth (1-based) command that starts with firstpart, or
None."""
        count = 0
        for c in reversed(self._commands):
            if c.startswith(firstpart):
                count += 1
                if count >= n:
                    return c
        return None
    def find_all(self, needle):
        """Find all commands that contain the given text. In order
        of being used.
        """
        commands = []
        for c in reversed(self._commands):
            if needle in c:
                commands.append(c)
        return commands
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Icons module
Defines functionality for creating icons by composing different overlays
and also by directly drawing into the pixmap. This allows making icons
that show information to the user in a very effective, yet subtle manner.
"""
from pyzo.qt import QtCore, QtGui, QtWidgets
import pyzo
class IconArtist:
    """IconArtist(icon=None)
    Object to draw icons with. Can be instantiated with an existing icon
    or as a blank icon. Perform operations and then use finish() to
    obtain the result.
    """
    def __init__(self, icon=None):
        # Get pixmap from given icon (None creates empty pixmap)
        self._pm = self._getPixmap(icon)
        # Instantiate painter for the pixmap
        self._painter = QtGui.QPainter()
        self._painter.begin(self._pm)
    def finish(self, icon=None):
        """finish()
        Finish the drawing and return the resulting icon.
        """
        self._painter.end()
        return QtGui.QIcon(self._pm)
    def _getPixmap(self, icon):
        # Get icon if given by name
        if isinstance(icon, str):
            icon = pyzo.icons[icon]
        # Create pixmap
        if icon is None:
            pm = QtGui.QPixmap(16, 16)
            pm.fill(QtGui.QColor(0, 0, 0, 0))
            return pm
        elif isinstance(icon, tuple):
            pm = QtGui.QPixmap(icon[0], icon[1])
            pm.fill(QtGui.QColor(0, 0, 0, 0))
            return pm
        elif isinstance(icon, QtGui.QPixmap):
            return icon
        elif isinstance(icon, QtGui.QIcon):
            return icon.pixmap(16, 16)
        else:
            raise ValueError("Icon for IconArtis should be icon, pixmap or
name.")
    def setPenColor(self, color):
        """setPenColor(color)
        Set the color of the pen. Color can be anything that can be passed to
        Qcolor().
        """
        pen = QtGui.QPen()
        if isinstance(color, tuple):
            pen.setColor(QtGui.QColor(*color))
        else:
            pen.setColor(QtGui.QColor(color))
        self._painter.setPen(pen)
    def addLayer(self, overlay, x=0, y=0):
        """addOverlay(overlay, x=0, y=0)
        Add an overlay icon to the icon (add the specified position).
        """
        pm = self._getPixmap(overlay)
        self._painter.drawPixmap(x, y, pm)
    def addLine(self, x1, y1, x2, y2):
        """addLine( x1, y1, x2, y2)
        Add a line to the icon.
        """
        self._painter.drawLine(x1, y1, x2, y2)
    def addPoint(self, x, y):
        """addPoint( x, y)
        Add a point to the icon.
        """
        self._painter.drawPoint(x, y)
    def addMenuArrow(self, strength=100):
        """addMenuArrow()
        Adds a menu arrow to the icon to let the user know the icon
        is clickable.
        """
        x, y = 0, 12
        a1, a2 = int(strength / 2), strength
        # Zeroth line of 3+2
        self.setPenColor((0, 0, 0, a1))
        self.addPoint(x + 0, y - 1)
        self.addPoint(x + 4, y - 1)
        self.setPenColor((0, 0, 0, a2))
        self.addPoint(x + 1, y - 1)
        self.addPoint(x + 2, y - 1)
        self.addPoint(x + 3, y - 1)
        # First line of 3+2
        self.setPenColor((0, 0, 0, a1))
        self.addPoint(x + 0, y + 0)
        self.addPoint(x + 4, y + 0)
        self.setPenColor((0, 0, 0, a2))
        self.addPoint(x + 1, y + 0)
        self.addPoint(x + 2, y + 0)
        self.addPoint(x + 3, y + 0)
        # Second line of 3
        self.addPoint(x + 1, y + 1)
        self.addPoint(x + 2, y + 1)
        self.addPoint(x + 3, y + 1)
        # Third line of 1+2
        self.addPoint(x + 2, y + 2)
        self.setPenColor((0, 0, 0, a1))
        self.addPoint(x + 1, y + 2)
        self.addPoint(x + 3, y + 2)
        # Fourth line of 1
        self.setPenColor((0, 0, 0, a2))
        self.addPoint(x + 2, y + 3)
# todo: not used; remove me?
class TabCloseButton(QtWidgets.QToolButton):
    """TabCloseButton
    This class implements a very compact close button to be used in tabs.
    It allows managing tab (the closing part of it) in a fast and intuitive
    fashion.
    """
    SIZE = 5, 8
    def __init__(self):
        QtWidgets.QToolButton.__init__(self)
        # Init
        self.setIconSize(QtCore.QSize(*self.SIZE))
        self.setStyleSheet("QToolButton{ border:none; padding:0px; margin:0px;
}")
        self.setIcon(self.getCrossIcon1())
    def mousePressEvent(self, event):
        # Get tabs
        tabs = self.parent().parent()
        # Get index from position
        pos = self.mapTo(tabs, event.pos())
        index = tabs.tabBar().tabAt(pos)
        # Close it
        tabs.tabCloseRequested.emit(index)
    def enterEvent(self, event):
        QtWidgets.QToolButton.enterEvent(self, event)
        self.setIcon(self.getCrossIcon2())
    def leaveEvent(self, event):
        QtWidgets.QToolButton.leaveEvent(self, event)
        self.setIcon(self.getCrossIcon1())
    def _createCrossPixmap(self, alpha):
        artist = IconArtist(self.SIZE)
        #
        artist.setPenColor((0, 0, 0, alpha))
        #
        artist.addPoint(0, 0)
        artist.addPoint(1, 1)
        artist.addPoint(2, 2)
        artist.addPoint(3, 3)
        artist.addPoint(4, 4)
        artist.addPoint(0, 4)
        artist.addPoint(1, 3)
        artist.addPoint(3, 1)
        artist.addPoint(4, 0)
        #
        artist.setPenColor((0, 0, 0, int(0.5 * alpha)))
        #
        artist.addPoint(1, 0)
        artist.addPoint(0, 1)
        artist.addPoint(2, 1)
        artist.addPoint(1, 2)
        artist.addPoint(3, 2)
        artist.addPoint(2, 3)
        artist.addPoint(4, 3)
        artist.addPoint(3, 4)
        #
        artist.addPoint(0, 3)
        artist.addPoint(1, 4)
        artist.addPoint(3, 0)
        artist.addPoint(4, 1)
        #
        return artist.finish().pixmap(*self.SIZE)
    def getCrossIcon1(self):
        if hasattr(self, "_cross1"):
            pm = self._cross1
        else:
            pm = self._createCrossPixmap(80)
        return QtGui.QIcon(pm)
    def getCrossIcon2(self):
        if hasattr(self, "_cross2"):
            pm = self._cross2
        else:
            pm = self._createCrossPixmap(240)
        # Set
        return QtGui.QIcon(pm)
# todo: not used; remove me?
class ToolButtonWithMenuIndication(QtWidgets.QToolButton):
    """ToolButtonWithMenuIndication
    Tool button that wraps the icon in a slightly larger icon that
    contains a small arrow that lights up when hovering over the icon.
    The button itself is not drawn. If the icon is clicked, the
    customContextMenuRequested signal of the "grandparent" is emitted. In
    this way we realize a suble icon that can be clicked on to show a menu.
    """
    SIZE = 21, 16
    def __init__(self):
        QtWidgets.QToolButton.__init__(self)
        # Init
        self.setIconSize(QtCore.QSize(*self.SIZE))
        self.setStyleSheet("QToolButton{ border: none; }")
        # Create arrow pixmaps
        self._menuarrow1 = self._createMenuArrowPixmap(0)
        self._menuarrow2 = self._createMenuArrowPixmap(70)
        self._menuarrow = self._menuarrow1
        # Variable to keep icon
        self._icon = None
        # Variable to keep track of when the mouse was pressed, so that
        # we can allow dragging as well as clicking the menu.
        self._menuPressed = False
    def mousePressEvent(self, event):
        # Ignore event so that the tabbar will change to that tab
        event.ignore()
        self._menuPressed = event.pos()
    def mouseMoveEvent(self, event):
        QtWidgets.QToolButton.mouseMoveEvent(self, event)
        if self._menuPressed:
            dragDist = QtWidgets.QApplication.startDragDistance()
            if (event.pos() - self._menuPressed).manhattanLength() >= dragDist:
                self._menuPressed = False
    def mouseReleaseEvent(self, event):
        event.ignore()
        if self._menuPressed:
            tabs = self.parent().parent()
            pos = self.mapTo(tabs, event.pos())
            tabs.customContextMenuRequested.emit(pos)
    def enterEvent(self, event):
        QtWidgets.QToolButton.enterEvent(self, event)
        self._menuarrow = self._menuarrow2
        self.setIcon()
        self._menuPressed = False
    def leaveEvent(self, event):
        QtWidgets.QToolButton.leaveEvent(self, event)
        self._menuarrow = self._menuarrow1
        self.setIcon()
        self._menuPressed = False
    def setIcon(self, icon=None):
        # Store icon if given, otherwise use buffered version
        if icon is not None:
            self._icon = icon
        # Compose icon by superimposing the menuarrow pixmap
        artist = IconArtist(self.SIZE)
        if self._icon:
            artist.addLayer(self._icon, 5, 0)
        artist.addLayer(self._menuarrow, 0, 0)
        icon = artist.finish()
        # Set icon
        QtWidgets.QToolButton.setIcon(self, icon)
    def _createMenuArrowPixmap(self, strength):
        artist = IconArtist()
        artist.addMenuArrow(strength)
        return artist.finish().pixmap(16, 16)
class TabToolButton(QtWidgets.QToolButton):
    """TabToolButton
    Base menu for editor and shell tabs.
    """
    SIZE = 16, 16
    def __init__(self, *args):
        QtWidgets.QToolButton.__init__(self, *args)
        # Init
        self.setIconSize(QtCore.QSize(*self.SIZE))
        self.setStyleSheet("QToolButton{ border: none; }")
    def mousePressEvent(self, event):
        # Ignore event so that the tabbar will change to that tab
        event.ignore()
class TabToolButtonWithCloseButton(TabToolButton):
    """TabToolButtonWithCloseButton
    Tool button that wraps the icon in a slightly larger icon that
    contains a small cross that can be used to invoke a close request.
    """
    SIZE = 22, 16
    CROSS_OFFSET = 0, 2
    def __init__(self, *args):
        TabToolButton.__init__(self, *args)
        # Variable to keep icon
        self._icon = None
        self._cross = self.getCrossPixmap1()
        # For mouse tracking inside icon
        self.setMouseTracking(True)
        self._overCross = False
    def _isOverCross(self, pos):
        x1, x2 = self.CROSS_OFFSET[0], self.CROSS_OFFSET[0] + 5 + 1
        y1, y2 = self.CROSS_OFFSET[1], self.CROSS_OFFSET[1] + 5 + 1
        if pos.x() >= x1 and pos.x() <= x2 and pos.y() >= y1 and pos.y() <= y2:
            return True
        else:
            return False
    def mousePressEvent(self, event):
        if self._isOverCross(event.pos()):
            # Accept event so that the tabbar will NOT change to that tab
            event.accept()
        else:
            event.ignore()
    def mouseReleaseEvent(self, event):
        if self._isOverCross(event.pos()):
            event.accept()
            # Get tabs
            tabs = self.parent().parent()
            # Get index from position
            pos = self.mapTo(tabs, event.pos())
            index = tabs.tabBar().tabAt(pos)
            # Close it
            tabs.tabCloseRequested.emit(index)
        else:
            event.ignore()
    def mouseMoveEvent(self, event):
        QtWidgets.QToolButton.mouseMoveEvent(self, event)
        new_overCross = self._isOverCross(event.pos())
        if new_overCross != self._overCross:
            self._overCross = new_overCross
            if new_overCross:
                self._cross = self.getCrossPixmap2()
            else:
                self._cross = self.getCrossPixmap1()
            self.setIcon()
    def leaveEvent(self, event):
        if self._overCross:
            self._overCross = False
            self._cross = self.getCrossPixmap1()
            self.setIcon()
    def setIcon(self, icon=None):
        # Store icon if given, otherwise use buffered version
        if icon is not None:
            self._icon = icon
        # Compose icon by superimposing the menuarrow pixmap
        artist = IconArtist(self.SIZE)
        if self._icon:
            if self.CROSS_OFFSET[0] > 8:
                artist.addLayer(self._icon, 0, 0)
            else:
                artist.addLayer(self._icon, 6, 0)
        artist.addLayer(self._cross, *self.CROSS_OFFSET)
        icon = artist.finish()
        # Set icon
        QtWidgets.QToolButton.setIcon(self, icon)
    def _createMenuArrowPixmap(self, strength):
        artist = IconArtist()
        artist.addMenuArrow(strength)
        return artist.finish().pixmap(16, 16)
    def _createCrossPixmap(self, alpha):
        artist = IconArtist((5, 5))
        #
        artist.setPenColor((0, 0, 0, alpha))
        #
        artist.addPoint(0, 0)
        artist.addPoint(1, 1)
        artist.addPoint(2, 2)
        artist.addPoint(3, 3)
        artist.addPoint(4, 4)
        artist.addPoint(0, 4)
        artist.addPoint(1, 3)
        artist.addPoint(3, 1)
        artist.addPoint(4, 0)
        #
        artist.setPenColor((0, 0, 0, int(0.5 * alpha)))
        #
        artist.addPoint(1, 0)
        artist.addPoint(0, 1)
        artist.addPoint(2, 1)
        artist.addPoint(1, 2)
        artist.addPoint(3, 2)
        artist.addPoint(2, 3)
        artist.addPoint(4, 3)
        artist.addPoint(3, 4)
        #
        artist.addPoint(0, 3)
        artist.addPoint(1, 4)
        artist.addPoint(3, 0)
        artist.addPoint(4, 1)
        #
        return artist.finish().pixmap(5, 5)
    def getCrossPixmap1(self):
        if hasattr(self, "_cross1"):
            pm = self._cross1
        else:
            pm = self._createCrossPixmap(50)
        return pm
    def getCrossPixmap2(self):
        if hasattr(self, "_cross2"):
            pm = self._cross2
        else:
            pm = self._createCrossPixmap(240)
        # Set
        return pm
class EditorTabToolButton(TabToolButtonWithCloseButton):
    """Button for the tabs of the editors. This is just a
    tight wrapper for the icon.
    """
    def updateIcon(self, isDirty, isMain, isPinned, nBlocks=10001):
        # Init drawing
        artist = IconArtist()
        # Create base
        if isDirty:
            artist.addLayer("page_white_dirty")
            artist.setPenColor("#f00")
        else:
            artist.addLayer("page_white")
            artist.setPenColor("#444")
        # Paint lines
        if not nBlocks:
            nLines = 0
        elif nBlocks <= 10:
            nLines = 1
        elif nBlocks <= 100:
            nLines = 2
        elif nBlocks <= 1000:
            nLines = 3
        elif nBlocks <= 10000:
            nLines = 4
        else:
            nLines = 5
        #
        fraction = float(nBlocks) / 10**nLines
        fraction = min(fraction, 1.0)
        #
        for i in range(nLines):
            y = 4 + 2 * i
            n = 5
            if y > 6:
                n = 8
            # if i == nLines-1:
            #    n = int(fraction * n)
            artist.addLine(4, y, 4 + n, y)
        # Overlays
        if isMain:
            artist.addLayer("overlay_star")
        if isPinned:
            artist.addLayer("overlay_thumbnail")
        if isDirty:
            artist.addLayer("overlay_disk")
        # Apply
        self.setIcon(artist.finish())
class ShellIconMaker:
    """Object that can make an icon for the shells"""
    POSITION = (6, 7)  # absolute position of center of wheel.
    # Relative position for the wheel at two levels. Center is at (3,,3)
    POSITIONS1 = [(2, 2), (3, 2), (4, 2), (4, 3), (4, 4), (3, 4), (2, 4), (2,
3)]
    POSITIONS2 = [
        (2, 1),
        (3, 1),
        (4, 1),
        (5, 2),
        (5, 3),
        (5, 4),
        (4, 5),
        (3, 5),
        (2, 5),
        (1, 4),
        (1, 3),
        (1, 2),
    ]
    # Maps to make transitions between levels more natural
    MAP1to2 = [1, 2, 4, 5, 7, 8, 10, 11]
    MAP2to1 = [1, 2, 3, 3, 4, 5, 5, 6, 7, 7, 0, 1]
    MAX_ITERS_IN_LEVEL_1 = 2
    def __init__(self, objectWithIcon):
        self._objectWithIcon = objectWithIcon
        # Motion properties
        self._index = 0
        self._level = 0
        self._count = 0  #  to count number of iters in level 1
        # Prepare blob pixmap
        self._blob = self._createBlobPixmap()
        self._legs = self._createLegsPixmap()
        # Create timer
        self._timer = QtCore.QTimer(None)
        self._timer.setInterval(200)
        self._timer.setSingleShot(False)
        self._timer.timeout.connect(self.onTimer)
    def setIcon(self, icon):
        self._objectWithIcon.setIcon(icon)
    def _createBlobPixmap(self):
        artist = IconArtist()
        artist.setPenColor((0, 150, 0, 255))
        artist.addPoint(1, 1)
        artist.setPenColor((0, 150, 0, 200))
        artist.addPoint(1, 0)
        artist.addPoint(1, 2)
        artist.addPoint(0, 1)
        artist.addPoint(2, 1)
        artist.setPenColor((0, 150, 0, 100))
        artist.addPoint(0, 0)
        artist.addPoint(2, 0)
        artist.addPoint(0, 2)
        artist.addPoint(2, 2)
        return artist.finish().pixmap(16, 16)
    def _createLegsPixmap(self):
        artist = IconArtist()
        x, y = self.POSITION
        artist.setPenColor((0, 50, 0, 150))
        artist.addPoint(x + 1, y - 1)
        artist.addPoint(x + 1, y - 2)
        artist.addPoint(x + 0, y - 2)
        artist.addPoint(x + 3, y + 1)
        artist.addPoint(x + 4, y + 1)
        artist.addPoint(x + 4, y + 2)
        artist.addPoint(x + 2, y + 3)
        artist.addPoint(x + 2, y + 4)
        artist.addPoint(x + 0, y + 3)
        artist.addPoint(x + 0, y + 4)
        artist.addPoint(x - 1, y + 2)
        artist.addPoint(x - 2, y + 2)
        artist.addPoint(x - 1, y + 0)
        artist.addPoint(x - 2, y + 0)
        return artist.finish().pixmap(16, 16)
    def updateIcon(self, status="Ready"):
        """updateIcon(status)
        Public method to set what state the icon must show.
        """
        # Normalize and store
        if isinstance(status, str):
            status = status.lower()
        self._status = status
        # Handle
        if status == "busy":
            self._index = 0
            if self._level == 2:
                self._index = self.MAP2to1[self._index]
            self._level = 1
        elif status == "very busy":
            self._index = 0
            if self._level == 1:
                self._index = self.MAP1to2[self._index]
            self._level = 2
        else:
            self._level = 0
        # At least one timer iteration
        self._timer.start()
    def _nextIndex(self):
        self._index += 1
        if self._level == 1 and self._index >= 8:
            self._index = 0
        elif self._level == 2 and self._index >= 12:
            self._index = 0
    def _index1(self):
        return self._index
    def _index2(self):
        n = [0, 8, 12][self._level]
        index = self._index + n / 2
        if index >= n:
            index -= n
        return int(index)
    def onTimer(self):
        """onTimer()
        Invoked on each timer iteration. Will call the static drawing
        methods if in level 0. Otherwise will invoke drawInMotion().
        This method also checks if we should change levels and calculates
        how this is best achieved.
        """
        if self._level == 0:
            # Turn of timer
            self._timer.stop()
            # Draw
            if self._status in ["ready", "more"]:
                self.drawReady()
            elif self._status == "debug":
                self.drawDebug()
            elif self._status == "dead":
                self.drawDead()
            else:
                self.drawDead()
        elif self._level == 1:
            # Draw
            self.drawInMotion()
            # Next, this is always intermediate
            self._nextIndex()
            self._count += 1
        elif self._level == 2:
            # Draw
            self.drawInMotion()
            # Next
            self._nextIndex()
    def drawReady(self):
        """drawReady()
        Draw static icon for when in ready mode.
        """
        artist = IconArtist("application")
        artist.addLayer(self._blob, *self.POSITION)
        self.setIcon(artist.finish())
    def drawDebug(self):
        """drawDebug()
        Draw static icon for when in debug mode.
        """
        artist = IconArtist("application")
        artist.addLayer(self._blob, *self.POSITION)
        artist.addLayer(self._legs)
        self.setIcon(artist.finish())
    def drawDead(self):
        """drawDead()
        Draw static empty icon for when the kernel is dead.
        """
        artist = IconArtist("application")
        self.setIcon(artist.finish())
    def drawInMotion(self):
        """drawInMotion()
        Draw one frame of the icon in motion. Position of the blobs
        is determined from the index and the list of locations.
        """
        # Init drawing
        artist = IconArtist("application")
        # Define params
        dx, dy = self.POSITION[0] - 3, self.POSITION[1] - 3
        blob = self._blob
        #
        if self._level == 1:
            positions = self.POSITIONS1
        elif self._level == 2:
            positions = self.POSITIONS2
        # Draw
        pos1 = positions[self._index1()]
        pos2 = positions[self._index2()]
        artist.addLayer(blob, pos1[0] + dx, pos1[1] + dy)
        artist.addLayer(blob, pos2[0] + dx, pos2[1] + dy)
        # Done
        self.setIcon(artist.finish())
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module kernelBroker
This module implements the interface between Pyzo and the kernel.
"""
import os, sys, time
import subprocess
import signal
import threading
import ctypes
import yoton
import pyzo  # local Pyzo (can be on a different box than where the user is)
from pyzo.util import zon as ssdf  # zon is ssdf-light
# To allow interpreters relative to (frozen) Pyzo app
EXE_DIR = os.path.abspath(os.path.dirname(sys.executable))
if EXE_DIR.endswith(".app/Contents/MacOS"):
    EXE_DIR = os.path.dirname(EXE_DIR.rsplit(".app")[0])
# Important: the yoton event loop should run somehow!
class KernelInfo(ssdf.Struct):
    """KernelInfo
    Describes all information for a kernel. This class can be used at
    the IDE as well as the kernelbroker.
    This information goes a long way from the pyzo config file to the
    kernel. The list pyzo.config.shellConfigs2 contains the configs
    for all kernels. These objects are edited in-place by the
    shell config.
    The shell keeps a reference of the shell config used to start the
    kernel. On each restart all information is resend. In this way,
    if a user changes a setting in the shell config, it is updated
    when the shell restarts.
    The broker also keeps a copy of the shell config. In this way,
    the shell might send no config information (or only partially
    update the config information) on a restart. This is not so
    relevant now, but it can be when we are running multiple people
    on a single kernel, and there is only one user who has the
    original config.
    """
    def __init__(self, info=None):
        super().__init__()
        # ----- Fixed parameters that define a shell -----
        # scriptFile is used to define the mode. If given, we run in
        # script-mode. Otherwise we run in interactive mode.
        # The name of this shell config. Can be used to name the kernel
        self.name = "Python"
        # The executable. This can be '/usr/bin/python3.1' or
        # 'c:/program files/python2.6/python.exe', etc.
        self.exe = ""
        # The GUI toolkit to embed the event loop of.
        # Instantiate with a value that is settable
        self.gui = "Auto"
        # The Python path. Paths should be separated by newlines.
        # '$PYTHONPATH' is replaced by environment variable by broker
        self.pythonPath = ""
        # The path of the current project, the kernel will prepend this
        # to the sys.path. The broker could prepend to PYTHONPATH, but
        # in this way it is more explicit (kernel can tell the user that
        # the project path was prepended).
        self.projectPath = ""
        # The full filename of the script to run.
        # If given, the kernel should run in script-mode.
        # The kernel will check whether this file exists, and will
        # revert to interactive mode if it doesn't.
        self.scriptFile = ""
        # Interactive-mode only:
        # The initial directory. Only used for interactive-mode; in
        # script-mode the initial directory is the dir of the script.
        self.startDir = ""
        # The Startup script (only used for interactive-mode).
        # - Empty string means run nothing,
        # - Single line means file name
        # - multiple lines means source code.
        # - '$PYTHONSTARTUP' uses the code in that file. Broker replaces this.
        self.startupScript = ""
        # Additional command line arguments, set by the kernel
        self.argv = ""
        # Additional environment variables
        self.environ = ""
        # Load info from ssdf struct. Make sure they are all strings
        if info:
            # Get struct
            if isinstance(info, dict):
                s = info
            elif isinstance(info, str):
                s = ssdf.loads(info)
            else:
                raise ValueError(
                    "Kernel info should be a string or ssdf struct, not %s"
                    % str(type(info))
                )
            # Inject values
            for key in s:
                val = s[key]
                if not val:
                    val = ""
                self[key] = val
    def tostring(self):
        return ssdf.saves(self)
def getCommandFromKernelInfo(info, port):
    info = KernelInfo(info)
    # Apply default exe
    exe = info.exe or "python"
    if exe.startswith("."):
        exe = os.path.abspath(os.path.join(EXE_DIR, exe))
    # Correct path when it contains spaces
    if exe.count(" ") and exe[0] != '"':
        exe = '"{}"'.format(exe)
    # Get start script
    startScript = os.path.join(pyzo.pyzoDir, "pyzokernel", "start.py")
    startScript = '"{}"'.format(startScript)
    # Build command
    command = exe + " " + startScript + " " + str(port)
    # Done
    return command
def getEnvFromKernelInfo(info):
    info = KernelInfo(info)
    pythonPath = info.pythonPath
    # Set default pythonPath (replace only first occurrence of $PYTHONPATH
    ENV_PP = os.environ.get("PYTHONPATH", "")
    pythonPath = pythonPath.replace("$PYTHONPATH", "\n" + ENV_PP + "\n", 1)
    pythonPath = pythonPath.replace("$PYTHONPATH", "")
    # Split paths, allow newlines and os.pathsep
    for splitChar in "\n\r" + os.pathsep:
        pythonPath = pythonPath.replace(splitChar, "\n")
    pythonPaths = [p.strip() for p in pythonPath.split("\n") if p]
    # Recombine using the OS's path separator
    pythonPath = os.pathsep.join(pythonPaths)
    # Add entry to Pythopath, so that we can import yoton
    # Note: an empty entry might cause trouble if the start-directory is
    # somehow overriden (see issue 128).
    pythonPath = pyzo.pyzoDir + os.pathsep + pythonPath
    # Prepare environment, remove references to tk libraries,
    # since they're wrong when frozen. Python will insert the
    # correct ones if required.
    env = os.environ.copy()
    #
    env.pop("TK_LIBRARY", "")
    env.pop("TCL_LIBRARY", "")
    env["PYTHONPATH"] = pythonPath
    # Jython does not use PYTHONPATH but JYTHONPATH
    env["JYTHONPATH"] = pyzo.pyzoDir + os.pathsep + os.environ.get("JYTHONPATH",
"")
    env["TERM"] = "dumb"  # we have a "dumb" terminal (see #422)
    # PyInstaller uses DllDirectory, but this leaks down to subprocesses,
    # causing GUI integration with QT-based libs to fail. Turn it off now.
    # https://github.com/pyinstaller/pyinstaller/issues/3795
    if sys.platform.startswith("win32"):
        ctypes.windll.kernel32.SetDllDirectoryA(None)
    # PyInstaller prepends the root app dir to LD_LIBRARY_PATH (see #665)
    if getattr(sys, "frozen", False) and "LD_LIBRARY_PATH" in env:
        to_remove = os.path.normpath(os.path.dirname(sys.executable))
        paths = env["LD_LIBRARY_PATH"].split(os.pathsep)
        paths = [os.path.normpath(p) for p in paths]
        paths = [p for p in paths if p != to_remove]
        env["LD_LIBRARY_PATH"] = os.pathsep.join(paths)
    # Remove Qt plugin directories, because it breaks Qt integration for the
kernel
    # on several systems. E.g. QT_QPA_PLATFORM_PLUGIN_PATH, QT_PLUGIN_PATH
    if getattr(sys, "frozen", False):
        frozen_path = os.path.normpath(sys.prefix).lower()
        for key, val in list(env.items()):
            if key.startswith(("QT_", "QML2_")):
                if frozen_path in os.path.normpath(val).lower():
                    env.pop(key, None)
        env["PATH"] = os.pathsep.join(
            x
            for x in os.getenv("PATH", "").split(os.pathsep)
            if not os.path.normpath(x).lower().startswith(frozen_path)
        )
    # Add dirs specific to this Python interpreter. Especially important with
    # Miniconda/Anaconda on Windows, see issue #591
    prefix = os.path.dirname(info.exe)
    envdirs = ["", "Library/usr/bin", "Library/bin", "bin"]
    if sys.platform.startswith("win"):
        envdirs.extend([r"Scripts", r"Library\mingw-w64\bin",
r"Library\mingw-w32\bin"])
    curpath = env.get("PATH", "").strip(os.pathsep)
    env["PATH"] = (
        os.pathsep.join(os.path.join(prefix, d) for d in envdirs) + os.pathsep +
curpath
    )
    # Add environment variables specified in shell config
    for line in info.environ.splitlines():
        line = line.strip()
        if "=" in line:
            key, val = line.split("=", 1)
            if key:
                key, val = key.strip(), val.strip()
                env[key] = os.path.expandvars(val)
    # Done
    return env
class KernelBroker:
    """KernelBroker(info)
    This class functions as a broker between a kernel process and zero or
    more IDE's (clients).
    This class has a single context assosiated with it, that lives as long
    as this object. It is used to connect to a kernel process and to
    0 or more IDE's (clients). The kernel process can be "restarted", meaning
    that it is terminated and a new process started.
    The broker is cleaned up if there is no kernel process AND no connections.
    """
    def __init__(self, manager, info, name=""):
        self._manager = manager
        # Store info that defines the kernel
        self._originalInfo = KernelInfo(info)
        # Make a copy for the current version. This copy is re-created on
        # each restart
        self._info = ssdf.copy(self._originalInfo)
        # Store name (or should the name be defined in the info struct)
        self._name = name
        # Create context for the connection to the kernel and IDE's
        # This context is persistent (it stays as long as this KernelBroker
        # instance is alive).
        self._context = yoton.Context()
        self._kernelCon = None
        self._ctrl_broker = None
        # Create yoton-based timer
        self._timer = yoton.Timer(0.2, oneshot=False)
        self._timer.bind(self.mainLoopIter)
        # Kernel process and connection (these are replaced on restarting)
        self._reset()
        # For restarting after terminating
        self._pending_restart = None
    ## Startup and teardown
    def _create_channels(self):
        ct = self._context
        # Close any existing channels first
        self._context.close_channels()
        # Create stream channels.
        # Stdout is for the C-level stdout/stderr streams.
        self._strm_broker = yoton.PubChannel(ct, "strm-broker")
        self._strm_raw = yoton.PubChannel(ct, "strm-raw")
        self._strm_prompt = yoton.PubChannel(ct, "strm-prompt")
        # Create control channel so that the IDE can control restarting etc.
        self._ctrl_broker = yoton.SubChannel(ct, "ctrl-broker")
        # Status channel to pass startup parameters to the kernel
        self._stat_startup = yoton.StateChannel(ct, "stat-startup",
yoton.OBJECT)
        # We use the stat-interpreter to set the status to dead when kernel dies
        self._stat_interpreter = yoton.StateChannel(ct, "stat-interpreter")
        # Create introspect channel so we can interrupt and terminate
        self._reqp_introspect = yoton.ReqChannel(ct, "reqp-introspect")
    def _reset(self, destroy=False):
        """_reset(destroy=False)
        Reset state. if destroy, does a full clean up, closing the context
        and removing itself from the KernelManager's list.
        """
        # Close connection (it might be in a wait state if the process
        # failed to start)
        if self._kernelCon is not None:
            self._kernelCon.close()
        # Set process and kernel connection to None
        self._process = None
        self._kernelCon = None
        self._terminator = None
        self._streamReader = None
        if destroy:
            # Stop timer
            self._timer.unbind(self.mainLoopIter)
            self._timer.stop()
            self._timer = None
            # Clean up this kernelbroker instance
            L = self._manager._kernels
            while self in L:
                L.remove(self)
            # Remove references
            #
            if self._context is not None:
                self._context.close()
            self._context = None
            #
            self._strm_broker = None
            self._strm_raw = None
            self._stat_startup = None
            self._stat_interpreter = None
            self._strm_prompt = None
            #
            self._ctrl_broker = None
            self._reqp_introspect = None
    def startKernelIfConnected(self, timeout=10.0):
        """startKernelIfConnected(timout=10.0)
        Start the kernel as soon as there is a connection.
        """
        self._process = time.time() + timeout
        self._timer.start()
    def startKernel(self):
        """startKernel()
        Launch the kernel in a subprocess, and connect to it via the
        context and two Pypes.
        """
        # Create channels
        self._create_channels()
        # Create info dict
        info = {}
        for key in self._info:
            info[key] = self._info[key]
        # Send info stuff so that the kernel has access to the information
        self._stat_startup.send(info)
        # Get directory to start process in
        cwd = pyzo.pyzoDir
        # Host connection for the kernel to connect
        # (tries several port numbers, staring from 'PYZO')
        self._kernelCon = self._context.bind(
            "localhost:PYZO", max_tries=256, name="kernel"
        )
        # Get command to execute, and environment to use
        command = getCommandFromKernelInfo(self._info, self._kernelCon.port1)
        env = getEnvFromKernelInfo(self._info)
        # Wrap command in call to 'cmd'?
        if sys.platform.startswith("win"):
            # as the author from Pype writes:
            # if we don't run via a command shell, then either sometimes we
            # don't get wx GUIs, or sometimes we can't kill the subprocesses.
            # And I also see problems with Tk.
            # But we only use it if we are sure that cmd is available.
            # See pyzo issue #240
            try:
                subprocess.check_output('cmd /c "cd"', shell=True)
            except (IOError, subprocess.SubprocessError):
                pass  # Do not use cmd
            else:
                command = 'cmd /c "{}"'.format(command)
        # Start process
        self._process = subprocess.Popen(
            command,
            shell=True,
            env=env,
            cwd=cwd,
            stdin=subprocess.PIPE,  # Fixes issue 165
            stdout=subprocess.PIPE,
            stderr=subprocess.STDOUT,
        )
        # Set timeout for connection, i.e. after how much time of
        # unresponsive ness is the kernel found to be running extension code
        # Better set this before connecting
        self._kernelCon.timeout = 0.5
        # Bind to events
        self._kernelCon.closed.bind(self._onKernelConnectionClose)
        self._kernelCon.timedout.bind(self._onKernelTimedOut)
        # Create reader for stream
        self._streamReader = StreamReader(
            self._process, self._strm_raw, self._strm_broker
        )
        # Start streamreader and timer
        self._streamReader.start()
        self._timer.start()
        # Reset some variables
        self._pending_restart = None
    def hostConnectionForIDE(self, address="localhost"):
        """hostConnectionForIDE()
        Host a connection for an IDE to connect to. Returns the port to which
        the ide can connect.
        """
        c = self._context.bind(address + ":PYZO+256", max_tries=256, name="ide")
        return c.port1
    ## Callbacks
    def _onKernelTimedOut(self, c, timedout):
        """_onKernelTimedOut(c, timeout)
        The kernel timed out (i.e. did not send heartbeat messages for
        a while. It is probably running extension code.
        """
        if timedout:
            self._stat_interpreter.send("Very busy")
        else:
            self._stat_interpreter.send("Busy")
    def _onKernelConnectionClose(self, c, why):
        """_onKernelConnectionClose(c, why)
        Connection with kernel lost. Tell clients why.
        """
        # If we receive this event while the current kernel connection
        # is not the one that generated the event, ignore it.
        if self._kernelCon is not c:
            return
        # The only reasonable way that the connection
        # can be lost without the kernel closing, is if the yoton context
        # crashed or was stopped somehow. In both cases, we lost control,
        # and should put it down!
        if not self._terminator:
            self.terminate("because connecton was lost", "KILL", 0.5)
    def _onKernelDied(self, returncode=0):
        """_onKernelDied()
        Kernel process died. Clean up!
        """
        # If the kernel did not start yet, probably the command is invalid
        if self._kernelCon and self._kernelCon.is_waiting:
            msg = "The process failed to start (invalid command?)."
        elif not self.isTerminating():
            msg = "Kernel process exited."
        elif not self._terminator._prev_action:
            # We did not actually take any terminating action
            # This happens, because if the kernel is killed from outside,
            # _onKernelConnectionClose() triggers a terminate sequence
            # (but with a delay).
            # Note the "The" to be able to distinguish this case
            msg = "The kernel process exited."
        else:
            msg = self._terminator.getMessage("Kernel process")
        if self._context.connection_count:
            # Notify
            returncodeMsg = "\n%s (%s)\n\n" % (msg, str(returncode))
            self._strm_broker.send(returncodeMsg)
            # Empty prompt and signal dead
            self._strm_prompt.send("\b")
            self._stat_interpreter.send("Dead")
            self._context.flush()
        # Cleanup (get rid of kernel process references)
        self._reset()
        # Handle any pending action
        if self._pending_restart:
            self.startKernel()
    ## Main loop and termination
    def terminate(self, reason="by user", action="TERM", timeout=0.0):
        """terminate(reason='by user', action='TERM', timeout=0.0)
        Initiate termination procedure for the current kernel.
        """
        # The terminatation procedure is started by creating
        # a KernelTerminator instance. This instance's iteration method
        # iscalled from _mailLoopIter().
        self._terminator = KernelTerminator(self, reason, action, timeout)
    def isTerminating(self):
        """isTerminating()
        Get whether the termination procedure has been initiated. This
        simply checks whether there is a self._terminator instance.
        """
        return bool(self._terminator)
    def mainLoopIter(self):
        """mainLoopIter()
        Periodically called. Kind of the main loop iteration for this kernel.
        """
        # Get some important status info
        hasProcess = self._process is not None
        hasKernelConnection = bool(self._kernelCon and
self._kernelCon.is_connected)
        hasClients = False
        if self._context:
            hasClients = self._context.connection_count >
int(hasKernelConnection)
        # Should we clean the whole thing up?
        if not (hasProcess or hasClients):
            self._reset(True)  # Also unregisters this timer callback
            return
        # Waiting to get started; waiting for client to connect
        if isinstance(self._process, float):
            if self._context.connection_count:
                self.startKernel()
            elif self._process > time.time():
                self._process = None
            return
        # If we have a process ...
        if self._process:
            # Test if process is dead
            process_returncode = self._process.poll()
            if process_returncode is not None:
                self._onKernelDied(process_returncode)
                return
            # Are we in the process of terminating?
            elif self.isTerminating():
                self._terminator.next()
        elif self.isTerminating():
            # We cannot have a terminator if we have no process
            self._terminator = None
        # handle control messages
        if self._ctrl_broker:
            for msg in self._ctrl_broker.recv_all():
                if msg == "INT":
                    self._commandInterrupt()
                elif msg == "TERM":
                    self._commandTerminate()
                elif msg.startswith("RESTART"):
                    self._commandRestart(msg)
                else:
                    pass  # Message is not for us
    def _commandInterrupt(self):
        if self._process is None:
            self._strm_broker.send("Cannot interrupt: process is dead.\n")
        # Kernel receives and acts
        elif sys.platform.startswith("win"):
            self._reqp_introspect.interrupt()
        else:
            # Use POSIX to interrupt, which is more reliable
            # (the introspect thread might not get a chance)
            # but also does not work if running extension code
            pid = self._kernelCon.pid2
            os.kill(pid, signal.SIGINT)
    def _commandTerminate(self):
        # Start termination procedure
        # Kernel will receive term and act (if it can).
        # If it wont, we will act in a second or so.
        if self._process is None:
            self._strm_broker.send("Cannot terminate: process is dead.\n")
        elif self.isTerminating():
            # The user gave kill command while the kill process
            # is running. We could do an immediate kill now,
            # or we let the terminate process run its course.
            pass
        else:
            self.terminate("by user")
    def _commandRestart(self, msg):
        # Almost the same as terminate, but now we have a pending action
        self._pending_restart = True
        # Recreate the info struct
        self._info = ssdf.copy(self._originalInfo)
        # Update the info struct
        new_info = ssdf.loads(msg.split("RESTART", 1)[1])
        for key in new_info:
            self._info[key] = new_info[key]
        # Restart now, wait, or initiate termination procedure?
        if self._process is None:
            self.startKernel()
        elif self.isTerminating():
            pass  # Already terminating
        else:
            self.terminate("for restart")
class KernelTerminator:
    """KernelTerminator(broker, reason='user terminated', action='TERM',
timeout=0.0)
    Simple class to help terminating the kernel. It has a next() method
    that should be periodically called. It keeps track whether the timeout
    has passed and will undertake increaslingly ruder actions to terminate
    the kernel.
    """
    def __init__(self, broker, reason="by user", action="TERM", timeout=0.0):
        # Init/store
        self._broker = broker
        self._reason = reason
        self._next_action = ""
        # Go
        self._do(action, timeout)
    def _do(self, action, timeout):
        self._prev_action = self._next_action
        self._next_action = action
        self._timeout = time.time() + timeout
        if not timeout:
            self.next()
    def next(self):
        # Get action
        action = self._next_action
        if time.time() < self._timeout:
            # Time did not pass yet
            pass
        elif action == "TERM":
            self._broker._reqp_introspect.terminate()
            self._do("INT", 0.5)
        elif action == "INT":
            # Count
            if not hasattr(self, "_count"):
                self._count = 0
            self._count += 1
            # Handle
            if self._count < 5:
                self._broker._reqp_introspect.interrupt()
                self._do("INT", 0.1)
            else:
                self._do("KILL", 0)
        elif action == "KILL":
            # Get pid and signal
            pid = self._broker._kernelCon.pid2
            sigkill = signal.SIGTERM
            if hasattr(signal, "SIGKILL"):
                sigkill = signal.SIGKILL
            # Kill
            if hasattr(os, "kill"):
                os.kill(pid, sigkill)
            elif sys.platform.startswith("win"):
                kernel32 = ctypes.windll.kernel32
                handle = kernel32.OpenProcess(1, 0, pid)
                kernel32.TerminateProcess(handle, 0)
                # os.system("TASKKILL /PID " + str(pid) + " /F")
            # Set what we did
            self._do("NOTHING", 9999999999999999)
    def getMessage(self, what):
        # Get nice string of that
        D = {
            "": "exited",
            "TERM": "terminated",
            "INT": "terminated (after interrupting)",
            "KILL": "killed",
        }
        actionMsg = D.get(self._prev_action, "stopped for unknown reason")
        # Compile stop-string
        return "{} {} {}.".format(what, actionMsg, self._reason)
class StreamReader(threading.Thread):
    """StreamReader(process, channel)
    Reads stdout of process and send to a yoton channel.
    This needs to be done in a separate thread because reading from
    a PYPE blocks.
    """
    def __init__(self, process, strm_raw, strm_broker):
        threading.Thread.__init__(self)
        self._process = process
        self._strm_raw = strm_raw
        self._strm_broker = strm_broker
        self.deamon = True
        self._exit = False
    def stop(self, timeout=1.0):
        self._exit = True
        self.join(timeout)
    def run(self):
        while not self._exit:
            time.sleep(0.001)
            # Read any stdout/stderr messages and route them via yoton.
            msg = self._process.stdout.readline()  # <-- Blocks here
            if not isinstance(msg, str):
                msg = msg.decode("utf-8", "ignore")
            try:
                self._strm_raw.send(msg)
            except IOError:
                pass  # Channel is closed
            # Process dead?
            if not msg:  # or self._process.poll() is not None:
                break
        # self._strm_broker.send('streamreader exit\n')
class Kernelmanager:
    """Kernelmanager
    This class manages a set of kernels. These kernels run on the
    same machine as this broker. IDE's can ask which kernels are available
    and can connect to them via this broker.
    The Pyzo process runs an instance of this class that connects at
    localhost. At a later stage, we may make it possible to create
    a kernel-server at a remote machine.
    """
    def __init__(self, public=False):
        # Set whether other machines in this network may connect to our kernels
        self._public = public
        # Init list of kernels
        self._kernels = []
    def createKernel(self, info, name=None):
        """create_kernel(info, name=None)
        Create a new kernel. Returns the port number to connect to the
        broker's context.
        """
        # Set name if not given
        if not name:
            i = len(self._kernels) + 1
            name = "kernel %i" % i
        # Create kernel
        kernel = KernelBroker(self, info, name)
        self._kernels.append(kernel)
        # Host a connection for the ide
        port = kernel.hostConnectionForIDE()
        # Tell broker to start as soon as the IDE connects with the broker
        kernel.startKernelIfConnected()
        # Done
        return port
    def getKernelList(self):
        # Get info of each kernel as an ssdf struct
        infos = []
        for kernel in self._kernels:
            info = kernel._info
            info = ssdf.loads(info.tostring())
            info.name = kernel._name
            infos.append(info)
        # Done
        return infos
    def terminateAll(self):
        """terminateAll()
        Terminates all kernels. Required when shutting down Pyzo.
        When this function returns, all kernels will be terminated.
        """
        for kernel in [kernel for kernel in self._kernels]:
            # Try closing the process gently: by closing stdin
            terminator = KernelTerminator(kernel, "for closing down")
            # Terminate
            while (
                kernel._kernelCon
                and kernel._kernelCon.is_connected
                and kernel._process
                and (kernel._process.poll() is None)
            ):
                terminator.next()
                time.sleep(0.02)
                yoton.process_events(False)
            # Clean up
            kernel._reset(True)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module main
This module contains the main frame. Implements the main window.
Also adds some variables to the pyzo namespace, such as the callLater
function which is also defined here.
"""
import os, sys, time
import base64
from queue import Queue, Empty
import pyzo
from pyzo.core.icons import IconArtist
from pyzo.core import commandline
from pyzo.core.statusbar import StatusBar
from pyzo import qt
from pyzo.qt import QtCore, QtGui, QtWidgets
from pyzo.core.splash import SplashWidget
from pyzo.util import paths
from pyzo.util import zon as ssdf  # zon is ssdf-light
from pyzo import translate
class MainWindow(QtWidgets.QMainWindow):
    def __init__(self, parent=None, locale=None):
        QtWidgets.QMainWindow.__init__(self, parent)
        self._closeflag = 0  # Used during closing/restarting
        # Init window title and application icon
        # Set title to something nice. On Ubuntu 12.10 this text is what
        # is being shown at the fancy title bar (since it's not properly
        # updated)
        self.setMainTitle()
        loadAppIcons()
        self.setWindowIcon(pyzo.icon)
        # Restore window geometry before drawing for the first time,
        # such that the window is in the right place
        self.resize(800, 600)  # default size
        self.restoreGeometry()
        # Show splash screen (we need to set our color too)
        w = SplashWidget(self, distro="no distro")
        self.setCentralWidget(w)
        self.setStyleSheet("QMainWindow { background-color: #268bd2;}")
        # Show empty window and disable updates for a while
        self.show()
        self.paintNow()
        self.setUpdatesEnabled(False)
        # Determine timeout for showing splash screen
        splash_timeout = time.time() + 1.0
        # Set locale of main widget, so that qt strings are translated
        # in the right way
        if locale:
            self.setLocale(locale)
        # Store myself
        pyzo.main = self
        # Init dockwidget settings
        self.setTabPosition(QtCore.Qt.AllDockWidgetAreas,
QtWidgets.QTabWidget.South)
        self.setDockOptions(
            QtWidgets.QMainWindow.AllowTabbedDocks
            | QtWidgets.QMainWindow.AllowNestedDocks
            # |  QtWidgets.QMainWindow.AnimatedDocks
        )
        # Set window atrributes
        self.setAttribute(QtCore.Qt.WA_AlwaysShowToolTips, True)
        # Load icons and fonts
        loadIcons()
        loadFonts()
        # Set qt style and test success
        self.setQtStyle(None)  # None means init!
        # Hold the splash screen if needed
        while time.time() < splash_timeout:
            QtWidgets.qApp.sendPostedEvents()
            QtWidgets.qApp.processEvents()
            time.sleep(0.05)
        # Populate the window (imports more code)
        self._populate()
        # Revert to normal background, and enable updates
        self.setStyleSheet("")
        self.setUpdatesEnabled(True)
        # Restore window state, force updating, and restore again
        self.restoreState()
        self.paintNow()
        self.restoreState()
        # Present user with wizard if he/she is new.
        if False:  # pyzo.config.state.newUser:
            from pyzo.util.pyzowizard import PyzoWizard
            w = PyzoWizard(self)
            w.show()  # Use show() instead of exec_() so the user can interact
with pyzo
        # Create new shell config if there is None
        if not pyzo.config.shellConfigs2:
            from pyzo.core.kernelbroker import KernelInfo
            pyzo.config.shellConfigs2.append(KernelInfo())
        # Focus on editor
        e = pyzo.editors.getCurrentEditor()
        if e is not None:
            e.setFocus()
        # Handle any actions
        commandline.handle_cmd_args()
    # To force drawing ourselves
    def paintEvent(self, event):
        QtWidgets.QMainWindow.paintEvent(self, event)
        self._ispainted = True
    def paintNow(self):
        """Enforce a repaint and keep calling processEvents until
        we are repainted.
        """
        self._ispainted = False
        self.update()
        while not self._ispainted:
            QtWidgets.qApp.sendPostedEvents()
            QtWidgets.qApp.processEvents()
            time.sleep(0.01)
    def _populate(self):
        # Delayed imports
        from pyzo.core.editorTabs import EditorTabs
        from pyzo.core.shellStack import ShellStackWidget
        from pyzo.core import codeparser
        from pyzo.core.history import CommandHistory
        from pyzo.tools import ToolManager
        # Instantiate tool manager
        pyzo.toolManager = ToolManager()
        # Check to install conda now ...
        # from pyzo.util.bootstrapconda import check_for_conda_env
        # check_for_conda_env()
        # Instantiate and start source-code parser
        if pyzo.parser is None:
            pyzo.parser = codeparser.Parser()
            pyzo.parser.start()
        # Create editor stack and make the central widget
        pyzo.editors = EditorTabs(self)
        self.setCentralWidget(pyzo.editors)
        # Create floater for shell
        self._shellDock = dock = QtWidgets.QDockWidget(self)
        if pyzo.config.settings.allowFloatingShell:
            dock.setFeatures(dock.DockWidgetMovable | dock.DockWidgetFloatable)
        else:
            dock.setFeatures(dock.DockWidgetMovable)
        dock.setObjectName("shells")
        dock.setWindowTitle("Shells")
        self.addDockWidget(QtCore.Qt.RightDockWidgetArea, dock)
        # Create shell stack
        pyzo.shells = ShellStackWidget(self)
        dock.setWidget(pyzo.shells)
        # Initialize command history
        pyzo.command_history = CommandHistory("command_history.py")
        # Create the default shell when returning to the event queue
        callLater(pyzo.shells.addShell)
        # Create status bar
        self.setStatusBar(StatusBar())
        # show or hide
        self.statusBar().setVisible(pyzo.config.view.showStatusbar)
        # Create menu
        from pyzo.core import menu
        pyzo.keyMapper = menu.KeyMapper()
        menu.buildMenus(self.menuBar())
        # Add the context menu to the editor
        pyzo.editors.addContextMenu()
        pyzo.shells.addContextMenu()
        # Load tools
        if pyzo.config.state.newUser and not pyzo.config.state.loadedTools:
            pyzo.toolManager.loadTool("pyzosourcestructure")
            pyzo.toolManager.loadTool("pyzofilebrowser", "pyzosourcestructure")
        elif pyzo.config.state.loadedTools:
            for toolId in pyzo.config.state.loadedTools:
                pyzo.toolManager.loadTool(toolId)
    def setMainTitle(self, path=None):
        """Set the title of the main window, by giving a file path."""
        if not path:
            # Plain title
            title = "Interactive Editor for Python"
        else:
            # Title with a filename
            name = os.path.basename(path)
            if os.path.isfile(path):
                pass
            elif name == path:
                path = translate("main", "unsaved")
            else:
                pass  # We hope the given path is informative
            # Set title
            tmp = {
                "fileName": name,
                "filename": name,
                "name": name,
                "fullPath": path,
                "fullpath": path,
                "path": path,
            }
            title = pyzo.config.advanced.titleText.format(**tmp)
        # Set
        self.setWindowTitle(title)
    def saveWindowState(self):
        """Save:
        * which tools are loaded
        * geometry of the top level windows
        * layout of dockwidgets and toolbars
        """
        # Save tool list
        tools = pyzo.toolManager.getLoadedTools()
        pyzo.config.state.loadedTools = tools
        # Store window geometry
        geometry = self.saveGeometry()
        geometry = base64.encodebytes(geometry.data()).decode("ascii")
        pyzo.config.state.windowGeometry = geometry
        # Store window state
        state = self.saveState()
        state = base64.encodebytes(state.data()).decode("ascii")
        pyzo.config.state.windowState = state
    def restoreGeometry(self, value=None):
        # Restore window position and whether it is maximized
        if value is not None:
            return super().restoreGeometry(value)
        # No value give, try to get it from the config
        if pyzo.config.state.windowGeometry:
            try:
                geometry = pyzo.config.state.windowGeometry
                geometry = base64.decodebytes(geometry.encode("ascii"))
                self.restoreGeometry(geometry)
            except Exception as err:
                print("Could not restore window geomerty: " + str(err))
    def restoreState(self, value=None):
        # Restore layout of dock widgets and toolbars
        if value is not None:
            return super().restoreState(value)
        # No value give, try to get it from the config
        if pyzo.config.state.windowState:
            try:
                state = pyzo.config.state.windowState
                state = base64.decodebytes(state.encode("ascii"))
                self.restoreState(state)
            except Exception as err:
                print("Could not restore window state: " + str(err))
    def setQtStyle(self, stylename=None):
        """Set the style and the palette, based on the given style name.
        If stylename is None or not given will do some initialization.
        If bool(stylename) evaluates to False will use the default style
        for this system. Returns the QStyle instance.
        """
        if stylename is None:
            # Initialize
            # Get native pallette (used below)
            QtWidgets.qApp.nativePalette = QtWidgets.qApp.palette()
            # Obtain default style name
            pyzo.defaultQtStyleName = str(QtWidgets.qApp.style().objectName())
            # Other than gtk+ and mac, Fusion/Cleanlooks looks best (in my
opinion)
            if "gtk" in pyzo.defaultQtStyleName.lower():
                pass  # Use default style
            elif "macintosh" in pyzo.defaultQtStyleName.lower():
                pass  # Use default style
            elif qt.QT_VERSION > "5":
                pyzo.defaultQtStyleName = "Fusion"
            else:
                pyzo.defaultQtStyleName = "Cleanlooks"
            # Set style if there is no style yet
            if not pyzo.config.view.qtstyle:
                pyzo.config.view.qtstyle = pyzo.defaultQtStyleName
        # Init
        if not stylename:
            stylename = pyzo.config.view.qtstyle
        # Check if this style exist, set to default otherwise
        styleNames = [name.lower() for name in QtWidgets.QStyleFactory.keys()]
        if stylename.lower() not in styleNames:
            stylename = pyzo.defaultQtStyleName
        # Try changing the style
        qstyle = QtWidgets.qApp.setStyle(stylename)
        # Set palette
        if qstyle:
            QtWidgets.qApp.setPalette(QtWidgets.qApp.nativePalette)
        # Done
        return qstyle
    def closeEvent(self, event):
        """Override close event handler."""
        # Are we restaring?
        restarting = time.time() - self._closeflag < 1.0  # noqa: F841
        # Save settings
        pyzo.saveConfig()
        pyzo.command_history.save()
        # Stop command server
        commandline.stop_our_server()
        # Proceed with closing...
        result = pyzo.editors.closeAll()
        if not result:
            self._closeflag = False
            event.ignore()
            return
        else:
            self._closeflag = True
            # event.accept()  # Had to comment on Windows+py3.3 to prevent error
        # Proceed with closing shells
        pyzo.localKernelManager.terminateAll()
        for shell in pyzo.shells:
            shell._context.close()
        # The tools need to be explicitly closed to allow them to clean up
        for toolname in pyzo.toolManager.getLoadedTools():
            tool = pyzo.toolManager.getTool(toolname)
            if hasattr(tool, "cleanUp"):
                tool.cleanUp()
        # Stop all threads (this should really only be daemon threads)
        import threading
        for thread in threading.enumerate():
            if hasattr(thread, "stop"):
                try:
                    thread.stop(0.1)
                except Exception:
                    pass
        #         # Wait for threads to die ...
        #         # This should not be necessary, but I used it in the hope that
it
        #         # would prevent the segfault on Python3.3. It didn't.
        #         timeout = time.time() + 0.5
        #         while threading.activeCount() > 1 and time.time() < timeout:
        #             time.sleep(0.1)
        #         print('Number of threads alive:', threading.activeCount())
        # Proceed as normal
        QtWidgets.QMainWindow.closeEvent(self, event)
    def restart(self):
        """Restart Pyzo."""
        self._closeflag = time.time()
        # Close
        self.close()
        if self._closeflag:
            # Get args
            args = [arg for arg in sys.argv]
            if not paths.is_frozen():
                # Prepend the executable name (required on Linux)
                lastBit = os.path.basename(sys.executable)
                args.insert(0, lastBit)
            # When running from the pip entry point pyzo.exe ... (issue #641)
            if (
                len(args) == 2
                and args[0] == "python.exe"
                and not os.path.isfile(args[1])
            ):
                args = ["python.exe", "-m", "pyzo"]
            if sys.platform == "win32":
                # workaround for MSVCRT issue with spaces in arguments
                #     https://bugs.python.org/issue436259
                from subprocess import list2cmdline
                args = [list2cmdline([s]) for s in args]
            # Replace the process!
            os.execv(sys.executable, args)
    def createPopupMenu(self):
        # Init menu
        menu = QtWidgets.QMenu()
        # Insert two items
        for item in ["Editors", "Shells"]:
            action = menu.addAction(item)
            action.setCheckable(True)
            action.setChecked(True)
            action.setEnabled(False)
        # Insert tools
        for tool in pyzo.toolManager.loadToolInfo():
            action = menu.addAction(tool.name)
            action.setCheckable(True)
            action.setChecked(bool(tool.instance))
            action.menuLauncher = tool.menuLauncher
        # Show menu and process result
        a = menu.popup(QtGui.QCursor.pos())
        if a:
            a.menuLauncher(not a.menuLauncher(None))
def loadAppIcons():
    """loadAppIcons()
    Load the application iconsr.
    """
    # Get directory containing the icons
    appiconDir = os.path.join(pyzo.pyzoDir, "resources", "appicons")
    # Determine template for filename of the application icon-files.
    fnameT = "pyzologo{}.png"
    # Construct application icon. Include a range of resolutions. Note that
    # Qt somehow does not use the highest possible res on Linux/Gnome(?), even
    # the logo of qt-designer when alt-tabbing looks a bit ugly.
    pyzo.icon = QtGui.QIcon()
    for sze in [16, 32, 48, 64, 128, 256]:
        fname = os.path.join(appiconDir, fnameT.format(sze))
        if os.path.isfile(fname):
            pyzo.icon.addFile(fname, QtCore.QSize(sze, sze))
    # Set as application icon. This one is used as the default for all
    # windows of the application.
    QtWidgets.qApp.setWindowIcon(pyzo.icon)
    # Construct another icon to show when the current shell is busy
    artist = IconArtist(pyzo.icon)  # extracts the 16x16 version
    artist.setPenColor("#0B0")
    for x in range(11, 16):
        d = x - 11  # runs from 0 to 4
        artist.addLine(x, 6 + d, x, 15 - d)
    pm = artist.finish().pixmap(16, 16)
    #
    pyzo.iconRunning = QtGui.QIcon(pyzo.icon)
    pyzo.iconRunning.addPixmap(pm)  # Change only 16x16 icon
def loadIcons():
    """loadIcons()
    Load all icons in the icon dir.
    """
    # Get directory containing the icons
    iconDir = os.path.join(pyzo.pyzoDir, "resources", "icons")
    # Construct other icons
    dummyIcon = IconArtist().finish()
    pyzo.icons = ssdf.new()
    for fname in os.listdir(iconDir):
        if fname.endswith(".png"):
            try:
                # Short and full name
                name = fname.split(".")[0]
                name = name.replace("pyzo_", "")  # discart prefix
                ffname = os.path.join(iconDir, fname)
                # Create icon
                icon = QtGui.QIcon()
                icon.addFile(ffname, QtCore.QSize(16, 16))
                # Store
                pyzo.icons[name] = icon
            except Exception as err:
                pyzo.icons[name] = dummyIcon
                print("Could not load icon %s: %s" % (fname, str(err)))
def loadFonts():
    """loadFonts()
    Load all fonts that come with Pyzo.
    """
    import pyzo.codeeditor  # we need pyzo and codeeditor namespace here
    # Get directory containing the icons
    fontDir = os.path.join(pyzo.pyzoDir, "resources", "fonts")
    # Get database object
    db = QtGui.QFontDatabase  # static class
    # Set default font
    pyzo.codeeditor.Manager.setDefaultFontFamily("DejaVu Sans Mono")
    # Load fonts that are in the fonts directory
    if os.path.isdir(fontDir):
        for fname in os.listdir(fontDir):
            if "oblique" in fname.lower():  # issue #461
```

```
                continue
            if os.path.splitext(fname)[1].lower() in [".otf", ".ttf"]:
                try:
                    db.addApplicationFont(os.path.join(fontDir, fname))
                except Exception as err:
                    print("Could not load font %s: %s" % (fname, str(err)))
class _CallbackEventHandler(QtCore.QObject):
    """Helper class to provide the callLater function."""
    def __init__(self):
        QtCore.QObject.__init__(self)
        self.queue = Queue()
    def customEvent(self, event):
        while True:
            try:
                callback, args = self.queue.get_nowait()
            except Empty:
                break
            try:
                callback(*args)
            except Exception as why:
                print("callback failed: {}:\n{}".format(callback, why))
    def postEventWithCallback(self, callback, *args):
        self.queue.put((callback, args))
        QtWidgets.qApp.postEvent(self, QtCore.QEvent(QtCore.QEvent.User))
def callLater(callback, *args):
    """callLater(callback, *args)
    Post a callback to be called in the main thread.
    """
    _callbackEventHandler.postEventWithCallback(callback, *args)
# Create callback event handler instance and insert function in pyzo namespace
_callbackEventHandler = _CallbackEventHandler()
pyzo.callLater = callLater
_SCREENSHOT_CODE = """
import random
numerator = 4
def get_number():
    # todo: something appears to be broken here
    val = random.choice(range(10))
    return numerator / val
class Groceries(list):
    \"\"\" Overloaded list class.
    \"\"\"
    def append_defaults(self):
        spam = 'yum'
        pie = 3.14159
        self.extend([spam, pie])
class GroceriesPlus(Groceries):
    \"\"\" Groceries with surprises!
    \"\"\"
    def append_random(self):
        value = get_number()
        self.append(value)
# Create some groceries
g = GroceriesPlus()
g.append_defaults()
g.append_random()
"""
def screenshotExample(width=1244, height=700):
    e = pyzo.editors.newFile()
    e.editor.setPlainText(_SCREENSHOT_CODE)
    pyzo.main.resize(width, height)
def screenshot(countdown=5):
    QtCore.QTimer.singleShot(countdown * 1000, _screenshot)
def _screenshot():
    # Grab
    print("SNAP!")
    pix = QtGui.QPixmap.grabWindow(pyzo.main.winId())
    # pix = QtGui.QPixmap.grabWidget(pyzo.main)
    # Get name
    i = 1
    while i > 0:
        name = "pyzo_screen_%s_%02i.png" % (sys.platform, i)
        fname = os.path.join(os.path.expanduser("~"), name)
        if os.path.isfile(fname):
            i += 1
        else:
            i = -1
    # Save screenshot and a thumb
    pix.save(fname)
    thumb = pix.scaledToWidth(500, QtCore.Qt.SmoothTransformation)
    thumb.save(fname.replace("screen", "thumb"))
    print("Screenshot and thumb saved in", os.path.expanduser("~"))
pyzo.screenshot = screenshot
pyzo.screenshotExample = screenshotExample
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module menu
Implements a menu that can be edited very easily. Every menu item is
represented by a class. Also implements a dialog to change keyboard
shortcuts.
"""
import os, sys, shutil, re
import webbrowser
from datetime import datetime
from urllib.request import urlopen
import ast
import json
from pyzo.qt import QtCore, QtGui, QtWidgets
import pyzo
from pyzo.core.compactTabWidget import CompactTabWidget
from pyzo.core.themeEdit import EditColorDialog
from pyzo.core.pyzoLogging import print  # noqa
from pyzo.core.assistant import PyzoAssistant
from pyzo import translate
from pyzo.core.pdfExport import PdfExport
def buildMenus(menuBar):
    """
    Build all the menus
    """
    menus = [
        FileMenu(menuBar, translate("menu", "File")),
        EditMenu(menuBar, translate("menu", "Edit")),
        ViewMenu(menuBar, translate("menu", "View")),
        SettingsMenu(menuBar, translate("menu", "Settings")),
        ShellMenu(menuBar, translate("menu", "Shell")),
        RunMenu(menuBar, translate("menu", "Run")),
        ToolsMenu(menuBar, translate("menu", "Tools")),
        HelpMenu(menuBar, translate("menu", "Help")),
    ]
    menuBar._menumap = {}
    menuBar._menus = menus
    for menu in menuBar._menus:
        menuBar.addMenu(menu)
        menuName = menu.__class__.__name__.lower().split("menu")[0]
        menuBar._menumap[menuName] = menu
    # Enable tooltips
    def onHover(action):
        # This ugly bit of code makes sure that the tooltip is refreshed
        # (thus raised above the submenu). This happens only once and after
        # ths submenu has become visible.
        if action.menu():
            if not hasattr(menuBar, "_lastAction"):
                menuBar._lastAction = None
                menuBar._haveRaisedTooltip = False
            if action is menuBar._lastAction:
                if (not menuBar._haveRaisedTooltip) and
action.menu().isVisible():
                    QtWidgets.QToolTip.hideText()
                    menuBar._haveRaisedTooltip = True
            else:
                menuBar._lastAction = action
                menuBar._haveRaisedTooltip = False
        # Set tooltip
        tt = action.statusTip()
        if hasattr(action, "_shortcutsText"):
            tt = tt + " ({})".format(action._shortcutsText)  # Add shortcuts
text in it
        QtWidgets.QToolTip.showText(QtGui.QCursor.pos(), tt)
    if not hasattr(QtWidgets.QMenu, "setToolTipsVisible"):
        menuBar.hovered.connect(onHover)
# todo: put many settings in an advanced settings dialog:
# - autocomp use keywords
# - autocomp case sensitive
# - autocomp select chars
# - Default parser / indentation (width and tabsOrSpaces) / line endings
# - Shell wrapping to 80 columns?
# - number of lines in shell
# - more stuff from pyzo.config.advanced?
def getShortcut(fullName):
    """Given the full name or an action, get the shortcut
    from the pyzo.config.shortcuts2 dict. A tuple is returned
    representing the two shortcuts."""
    if isinstance(fullName, QtWidgets.QAction):
        fullName = fullName.menuPath  # the menuPath property is set in
Menu._addAction
    shortcut = "", ""
    if fullName in pyzo.config.shortcuts2:
        shortcut = pyzo.config.shortcuts2[fullName]
        if shortcut.count(","):
            shortcut = tuple(shortcut.split(","))
        else:
            shortcut = shortcut, ""
    return shortcut
def translateShortcutToOSNames(shortcut):
    """
    Translate Qt names to OS names (e.g. Ctrl -> cmd symbol for Mac,
    Meta -> Windows for windows
    """
    if sys.platform == "darwin":
        replace = (
            ("Ctrl+", "\u2318"),
            ("Shift+", "\u21E7"),
            ("Alt+", "\u2325"),
            ("Meta+", "^"),
        )
    else:
        replace = ()
    for old, new in replace:
        shortcut = shortcut.replace(old, new)
    return shortcut
class KeyMapper(QtCore.QObject):
    """
    This class is accessable via pyzo.keyMapper
    pyzo.keyMapper.keyMappingChanged is emitted when keybindings are changed
    """
    keyMappingChanged = QtCore.Signal()
    def setShortcut(self, action):
        """
        When an action is created or when keymappings are changed, this method
        is called to set the shortcut of an action based on its menuPath
        (which is the key in pyzo.config.shortcuts2, e.g. shell__clear_screen)
        """
        if action.menuPath in pyzo.config.shortcuts2:
            # Set shortcut so Qt can do its magic
            shortcuts = pyzo.config.shortcuts2[action.menuPath]
            action.setShortcuts(shortcuts.split(","))
            pyzo.main.addAction(
                action
            )  # issue #470, http://stackoverflow.com/questions/23916623
            # Also store shortcut text (used in display of tooltip
            shortcuts = shortcuts.replace(",", ", ").replace("  ", " ")
            action._shortcutsText = shortcuts.rstrip(", ")
def unwrapText(text):
    """Unwrap text to display in message boxes. This just removes all
    newlines. If you want to insert newlines, use \\r."""
    # Removes newlines
    text = text.replace("\n", "")
    # Remove double/triple/etc spaces
    text = text.lstrip()
    for i in range(10):
        text = text.replace("  ", " ")
    # Convert \\r newlines
    text = text.replace("\r", "\n")
    # Remove spaces after newlines
    text = text.replace("\n ", "\n")
    return text
class Menu(QtWidgets.QMenu):
    """Menu(parent=None, name=None)
    Base class for all menus. Has methods to add actions of all sorts.
    The add* methods all have the name and icon as first two arguments.
    This is not so consistent with the Qt API for addAction, but it allows
    for cleaner code to add items; the first item can be quite long because
    it is a translation. In the current API, the second and subsequent
    arguments usually fit nicely on the second line.
    """
    _dummyActionForHiddenEntryInKeyMapDialog = QtWidgets.QAction()
    def __init__(self, parent=None, name=None):
        QtWidgets.QMenu.__init__(self, parent)
        # Make sure that the menu has a title
        if name:
            self.setTitle(name)
        else:
            raise ValueError
        try:
            self.setToolTipsVisible(True)
        except Exception:
            pass
        # Set tooltip too.
        if hasattr(name, "tt"):
            self.setStatusTip(name.tt)
            self.setToolTip(name.tt)
        # Action groups within the menu keep track of the selected value
        self._groups = {}
        # menuPath is used to bind shortcuts, it is ,e.g. shell__clear_screen
        if hasattr(parent, "menuPath"):
            self.menuPath = parent.menuPath + "__"
        else:
            self.menuPath = ""  # This is a top-level menu
        # Get key for this menu
        key = name
        if hasattr(name, "key"):
            key = name.key
        self.menuPath += self._createMenuPathName(key)
        # Build the menu. Happens only once
        self.build()
    def _createMenuPathName(self, name):
        """
        Convert a menu title into a menuPath component name
        e.g. Interrupt current shell -> interrupt_current_shell
        """
        # hide anything between brackets
        name = re.sub("\(.*\)", "", name)
        # replace invalid chars
        name = name.replace(" ", "_")
        if name and name[0] in "0123456789_":
            name = "_" + name
        name = re.sub("[^a-zA-z_0-9]", "", name)
        return name.lower()
    def _addAction(self, text, icon, selected=None):
        """Convenience function that makes the right call to addAction()."""
        # Add the action
        if icon is None:
            a = self.addAction(text)
        else:
            a = self.addAction(icon, text)
        # Checkable?
        if selected is not None:
            a.setCheckable(True)
            a.setChecked(selected)
        # Set tooltip if we can find it
        if hasattr(text, "tt"):
            a.setStatusTip(text.tt)
            a.setToolTip(text.tt)
        # Find the key (untranslated name) for this menu item
        key = a.text()
        if hasattr(text, "key"):
            key = text.key
        a.menuPath = self.menuPath + "__" + self._createMenuPathName(key)
        # Register the action so its keymap is kept up to date
        pyzo.keyMapper.keyMappingChanged.connect(lambda:
pyzo.keyMapper.setShortcut(a))
        pyzo.keyMapper.setShortcut(a)
        return a
    def build(self):
        """
        Add all actions to the menu. To be overridden.
        """
        pass
    def popup(self, pos, action=None):
        self._pos = pos
        super().popup(pos, action)
    def addMenu(self, menu, icon=None):
        """
        Add a (sub)menu to this menu.
        """
        # Add menu in the conventional way
        a = QtWidgets.QMenu.addMenu(self, menu)
        a.menuPath = menu.menuPath
        # Set icon
        if icon is not None:
            a.setIcon(icon)
        return menu
    def addItem(self, text, icon=None, callback=None, value=None):
        """
        Add an item to the menu. If callback is given and not None,
        connect triggered signal to the callback. If value is None or not
        given, callback is called without parameteres, otherwise it is called
        with value as parameter
        """
        # Add action
        a = self._addAction(text, icon)
        # Connect the menu item to its callback
        if callback:
            if value is not None:
                a.triggered.connect(lambda b=None, v=value: callback(v))
            else:
                a.triggered.connect(lambda b=None: callback())
        return a
    def addGroupItem(self, text, icon=None, callback=None, value=None,
group=None):
        """
        Add a 'select-one' option to the menu. Items with equal group value form
        a group. If callback is specified and not None, the callback is called
        for the new active item, with the value for that item as parameter
        whenever the selection is changed
        """
        # Init action
        a = self._addAction(text, icon)
        a.setCheckable(True)
        # Connect the menu item to its callback (toggled is a signal only
        # emitted by checkable actions, and can also be called programmatically,
        # e.g. in QActionGroup)
        if callback:
            def doCallback(b, v):
                if b:
                    callback(v)
            a.toggled.connect(lambda b=None, v=value: doCallback(a.isChecked(),
v))
        # Add the menu item to a action group
        if group is None:
            group = "default"
        if group not in self._groups:
            # self._groups contains tuples (actiongroup, dict-of-actions)
            self._groups[group] = (QtWidgets.QActionGroup(self), {})
        actionGroup, actions = self._groups[group]
        actionGroup.addAction(a)
        actions[value] = a
        return a
    def addCheckItem(self, text, icon=None, callback=None, value=None,
selected=False):
        """
        Add a true/false item to the menu. If callback is specified and not
        None, the callback is called when the item is changed. If value is not
        specified or None, callback is called with the new state as parameter.
        Otherwise, it is called with the new state and value as parameters
        """
        # Add action
        a = self._addAction(text, icon, selected)
        # Connect the menu item to its callback
        if callback:
            if value is not None:
                a.triggered.connect(lambda b=None, v=value:
```

```
callback(a.isChecked(), v))
            else:
                a.triggered.connect(lambda b=None: callback(a.isChecked()))
        return a
    def setCheckedOption(self, group, value):
        """
        Set the selected value of a group. This will also activate the
        callback function of the item that gets selected.
        if group is None the default group is used.
        """
        if group is None:
            group = "default"
        actionGroup, actions = self._groups[group]
        if value in actions:
            actions[value].setChecked(True)
class GeneralOptionsMenu(Menu):
    """GeneralOptionsMenu(parent, name, callback, options=None)
    Menu to present the user with a list from which to select one item.
    We need this a lot.
    """
    def __init__(self, parent=None, name=None, callback=None, options=None):
        Menu.__init__(self, parent, name)
        self._options_callback = callback
        if options:
            self.setOptions(options)
    def build(self):
        pass  # We build when the options are given
    def setOptions(self, options, values=None):
        """
        Set the list of options, clearing any existing options. The options
        are added ad group items and registered to the callback given
        at initialization.
        """
        # Init
        self.clear()
        cb = self._options_callback
        # Get values
        if values is None:
            values = options
        for option, value in zip(options, values):
            self.addGroupItem(option, None, cb, value)
class IndentationMenu(Menu):
    """
    Menu for the user to control the type of indentation for a document:
    tabs vs spaces and the amount of spaces.
    Part of the File menu.
    """
    def build(self):
        self._items = [
            self.addGroupItem(
                translate("menu", "Use tabs"),
                None,
                self._setStyle,
                False,
                group="style",
            ),
            self.addGroupItem(
                translate("menu", "Use spaces"),
                None,
                self._setStyle,
                True,
                group="style",
            ),
        ]
        self.addSeparator()
        spaces = translate("menu", "spaces", "plural of spacebar character")
        self._items += [
            self.addGroupItem(
                "%d %s" % (i, spaces), None, self._setWidth, i, group="width"
            )
            for i in range(2, 9)
        ]
    def _setWidth(self, width):
        editor = pyzo.editors.getCurrentEditor()
        if editor is not None:
            editor.setIndentWidth(width)
    def _setStyle(self, style):
        editor = pyzo.editors.getCurrentEditor()
        if editor is not None:
            editor.setIndentUsingSpaces(style)
class FileMenu(Menu):
    def build(self):
        icons = pyzo.icons
        self._items = []
        # Create indent menu
        t = translate(
            "menu", "Indentation ::: The indentation used of the current file."
        )
        self._indentMenu = IndentationMenu(self, t)
        # Create parser menu
        from pyzo import codeeditor
        t = translate(
            "menu", "Syntax parser ::: The syntax parser of the current file."
        )
        self._parserMenu = GeneralOptionsMenu(self, t, self._setParser)
        self._parserMenu.setOptions(["Plain"] +
codeeditor.Manager.getParserNames())
        # Create line ending menu
        t = translate(
            "menu", "Line endings ::: The line ending character of the current
file."
        )
        self._lineEndingMenu = GeneralOptionsMenu(self, t, self._setLineEndings)
        self._lineEndingMenu.setOptions(["LF", "CR", "CRLF"])
        # Create encoding menu
        t = translate(
            "menu", "Encoding ::: The character encoding of the current file."
        )
        self._encodingMenu = GeneralOptionsMenu(self, t, self._setEncoding)
        # Bind to signal
        pyzo.editors.currentChanged.connect(self.onEditorsCurrentChanged)
        # Build menu file management stuff
        self.addItem(
            translate("menu", "New ::: Create a new (or temporary) file."),
            icons.page_add,
            pyzo.editors.newFile,
        )
        self.addItem(
            translate("menu", "Open... ::: Open an existing file from disk."),
            icons.folder_page,
            pyzo.editors.openFile,
        )
        #
        self._items += [
            self.addItem(
                translate("menu", "Save ::: Save the current file to disk."),
                icons.disk,
                pyzo.editors.saveFile,
            ),
            self.addItem(
                translate(
                    "menu", "Save as... ::: Save the current file under another 
name."
                ),
                icons.disk_as,
                pyzo.editors.saveFileAs,
            ),
            self.addItem(
                translate("menu", "Save all ::: Save all open files."),
                icons.disk_multiple,
                pyzo.editors.saveAllFiles,
            ),
            self.addItem(
                translate("menu", "Close ::: Close the current file."),
                icons.page_delete,
                pyzo.editors.closeFile,
            ),
            self.addItem(
                translate("menu", "Close all ::: Close all files."),
                icons.page_delete_all,
                pyzo.editors.closeAllFiles,
            ),
            self.addItem(
                translate(
                    "menu",
                    "Export to PDF ::: Export current file to PDF (e.g. for 
printing).",
                ),
                None,
                lambda: PdfExport().exec_(),
            ),
        ]
        # Build file properties stuff
        self.addSeparator()
        self._items += [
            self.addMenu(self._indentMenu, icons.page_white_gear),
            self.addMenu(self._parserMenu, icons.page_white_gear),
            self.addMenu(self._lineEndingMenu, icons.page_white_gear),
            self.addMenu(self._encodingMenu, icons.page_white_gear),
        ]
        # Closing of app
        self.addSeparator()
        self.addItem(
            translate("menu", "Restart Pyzo ::: Restart the application."),
            icons.arrow_rotate_clockwise,
            pyzo.main.restart,
        )
        self.addItem(
            translate("menu", "Quit Pyzo ::: Close the application."),
            icons.cancel,
            pyzo.main.close,
        )
        # Start disabled
        self.setEnabled(False)
    def setEnabled(self, enabled):
        """Enable or disable all items. If disabling, also uncheck all items"""
        for child in self._items:
            child.setEnabled(enabled)
    def onEditorsCurrentChanged(self):
        editor = pyzo.editors.getCurrentEditor()
        if editor is None:
            self.setEnabled(False)  # Disable / uncheck all editor-related
options
        else:
            self.setEnabled(True)
            # Update indentation
            self._indentMenu.setCheckedOption("style",
editor.indentUsingSpaces())
            self._indentMenu.setCheckedOption("width", editor.indentWidth())
            # Update parser
            parserName = "Plain"
            if editor.parser():
                parserName = editor.parser().name() or "Plain"
            self._parserMenu.setCheckedOption(None, parserName)
            # Update line ending
            self._lineEndingMenu.setCheckedOption(None,
editor.lineEndingsHumanReadable)
            # Update encoding
            self._updateEncoding(editor)
    def _setParser(self, value):
        editor = pyzo.editors.getCurrentEditor()
        if value.lower() == "plain":
            value = None
        if editor is not None:
            editor.setParser(value)
    def _setLineEndings(self, value):
        editor = pyzo.editors.getCurrentEditor()
        editor.lineEndings = value
    def _updateEncoding(self, editor):
        # Dict with encoding aliases (official to aliases)
        D = {
            "cp1250": ("windows-1252",),
            "cp1251": ("windows-1251",),
            "latin_1": ("iso-8859-1", "iso8859-1", "cp819", "latin", "latin1",
"L1"),
        }
        # Dict with aliases mapping to "official value"
        Da = {}
        for key in D:
            for key2 in D[key]:
                Da[key2] = key
        # Encodings to list
        encodings = ["utf-8", "ascii", "latin_1", "cp1250", "cp1251"]
        # Get current encoding (add if not present)
        editorEncoding = editor.encoding
        if editorEncoding in Da:
            editorEncoding = Da[editorEncoding]
        if editorEncoding not in encodings:
            encodings.append(editorEncoding)
        # Handle aliases
        encodingNames, encodingValues = [], []
        for encoding in encodings:
            encodingValues.append(encoding)
            if encoding in D:
                name = "%s (%s)" % (encoding, ", ".join(D[encoding]))
                encodingNames.append(name)
            else:
                encodingNames.append(encoding)
        # Update
        self._encodingMenu.setOptions(encodingNames, encodingValues)
        self._encodingMenu.setCheckedOption(None, editorEncoding)
    def _setEncoding(self, value):
        editor = pyzo.editors.getCurrentEditor()
        if editor is not None:
            editor.encoding = value
# todo: move to matching brace
class EditMenu(Menu):
    def build(self):
        icons = pyzo.icons
        self.addItem(
            translate("menu", "Undo ::: Undo the latest editing action."),
            icons.arrow_undo,
            self._editItemCallback,
            "undo",
        )
        self.addItem(
            translate("menu", "Redo ::: Redo the last undone editing action."),
            icons.arrow_redo,
            self._editItemCallback,
            "redo",
        )
        self.addSeparator()
        self.addItem(
            translate("menu", "Cut ::: Cut the selected text."),
            icons.cut,
            self._editItemCallback,
            "cut",
        )
        self.addItem(
            translate("menu", "Copy ::: Copy the selected text to the 
clipboard."),
            icons.page_white_copy,
            self._editItemCallback,
            "copy",
        )
        self.addItem(
            translate("menu", "Paste ::: Paste the text that is now on the 
clipboard."),
            icons.paste_plain,
            self._editItemCallback,
            "paste",
        )
        self.addItem(
            translate(
                "menu",
                "Paste and select ::: Paste the text that is now on the 
clipboard and keep it selected in order to change its indentation.",
            ),  # noqa
            icons.paste_plain,
            self._editItemCallback,
            "pasteAndSelect",
        )
        self.addItem(
            translate("menu", "Select all ::: Select all text."),
            icons.sum,
            self._editItemCallback,
            "selectAll",
        )
        self.addSeparator()
        self.addItem(
            translate("menu", "Indent ::: Indent the selected line."),
            icons.text_indent,
            self._editItemCallback,
            "indentSelection",
        )
        self.addItem(
            translate("menu", "Dedent ::: Unindent the selected line."),
            icons.text_indent_remove,
            self._editItemCallback,
            "dedentSelection",
        )
        self.addItem(
            translate("menu", "Comment ::: Comment the selected line."),
            icons.comment_add,
            self._editItemCallback,
            "commentCode",
        )
        self.addItem(
            translate("menu", "Uncomment ::: Uncomment the selected line."),
            icons.comment_delete,
            self._editItemCallback,
            "uncommentCode",
        )
        self.addItem(
            translate(
                "menu", "Toggle Comment ::: Toggle comment for the selected
line."
            ),
            None,
            self._editItemCallback,
            "toggleCommentCode",
        )
        self.addItem(
            translate(
                "menu",
                "Justify comment/docstring::: Reshape the selected text so it is
aligned to around 70 characters.",
            ),
            icons.text_align_justify,
            self._editItemCallback,
            "justifyText",
        )
        self.addItem(
            translate("menu", "Go to line ::: Go to a specific line number."),
            None,
            self._editItemCallback,
            "gotoLinePopup",
        )
        self.addItem(
            translate("menu", "Duplicate line ::: Duplicate the selected
line(s)."),
            None,
            self._editItemCallback,
            "duplicateLines",
        )
        self.addItem(
            translate("menu", "Delete line ::: Delete the selected line(s)."),
            None,
            self._editItemCallback,
            "deleteLines",
        )
        self.addSeparator()
        self.addItem(
            translate(
                "menu", "Toggle breakpoint ::: Toggle breakpoint on the current
line."
            ),
            None,
            self._editItemCallback,
            "toggleBreakpoint",
        )
        self.addSeparator()
        self.addItem(
            translate(
                "menu", "Toggle Case ::: Change selected text to lower or upper
case"
            ),
            None,
            self._editItemCallback,
            "toggleCase",
        )
        self.addSeparator()
        self.addItem(
            translate(
                "menu",
                "Find or replace ::: Show find/replace widget. Initialize with
selected text.",
            ),
            icons.find,
            pyzo.editors._findReplace.startFind,
        )
        self.addItem(
            translate(
                "menu",
                "Find selection ::: Find the next occurrence of the selected
text.",
            ),
            None,
            pyzo.editors._findReplace.findSelection,
        )
        self.addItem(
            translate(
                "menu",
                "Find selection backward ::: Find the previous occurrence of the
selected text.",
            ),
            None,
            pyzo.editors._findReplace.findSelectionBw,
        )
        self.addItem(
            translate(
                "menu", "Find next ::: Find the next occurrence of the search
string."
            ),
            None,
            pyzo.editors._findReplace.findNext,
        )
        self.addItem(
            translate(
                "menu",
                "Find previous ::: Find the previous occurrence of the search
string.",
            ),
            None,
            pyzo.editors._findReplace.findPrevious,
        )
    def _editItemCallback(self, action):
        widget = QtWidgets.qApp.focusWidget()
        # If the widget has a 'name' attribute, call it
        if hasattr(widget, action):
            getattr(widget, action)()
class ZoomMenu(Menu):
    """
    Small menu for the zooming. Part of the view menu.
    """
    def build(self):
        self.addItem(translate("menu", "Zoom in"), None, self._setZoom, +1)
        self.addItem(translate("menu", "Zoom out"), None, self._setZoom, -1)
        self.addItem(translate("menu", "Zoom reset"), None, self._setZoom, 0)
    def _setZoom(self, value):
        if not value:
            pyzo.config.view.zoom = 0
        else:
            pyzo.config.view.zoom += value
        # Apply
        for editor in pyzo.editors:
            pyzo.config.view.zoom = editor.setZoom(pyzo.config.view.zoom)
        for shell in pyzo.shells:
            pyzo.config.view.zoom = shell.setZoom(pyzo.config.view.zoom)
        logger = pyzo.toolManager.getTool("pyzologger")
        if logger:
            logger.updateZoom()
class FontMenu(Menu):
    _hide_in_key_map_dialog = True
    def __init__(self, parent=None, name="Font", *args, **kwds):
        Menu.__init__(self, parent, name, *args, **kwds)
        self.aboutToShow.connect(self._updateFonts)
    def _updateFonts(self):
        self.clear()
        # Build list with known available monospace fonts
        names = pyzo.codeeditor.Manager.fontNames()
        defaultName = "DejaVu Sans Mono"
        for name in sorted(names):
            default_suffix = " (%s)" % translate("menu", "default")
            txt = name + default_suffix if name == defaultName else name
            self.addGroupItem(txt, None, self._selectFont, value=name)
        # Select the current one
        self.setCheckedOption(None, pyzo.config.view.fontname)
    def _selectFont(self, name):
        pyzo.config.view.fontname = name
        # Apply
        for editor in pyzo.editors:
            editor.setFont(pyzo.config.view.fontname)
        for shell in pyzo.shells:
            shell.setFont(pyzo.config.view.fontname)
        logger = pyzo.toolManager.getTool("pyzologger")
        if logger:
            logger.updateFont()
# todo: brace matching
# todo: code folding?
# todo: maybe move qt theme to settings
class ViewMenu(Menu):
    def build(self):
        icons = pyzo.icons
        # Create edge column menu
        t = translate(
            "menu",
            "Location of long line indicator ::: The location of the long-line-
indicator.",
        )
        self._edgeColumMenu = GeneralOptionsMenu(self, t, self._setEdgeColumn)
        values = [0] + [i for i in range(60, 130, 10)]
        names = [translate("menu", "None")] + [str(i) for i in values[1:]]
        self._edgeColumMenu.setOptions(names, values)
        self._edgeColumMenu.setCheckedOption(None, pyzo.config.view.edgeColumn)
        # Create qt theme menu
        t = translate("menu", "Qt theme ::: The styling of the user interface
widgets.")
        self._qtThemeMenu = GeneralOptionsMenu(self, t, self._setQtTheme)
        styleNames = list(QtWidgets.QStyleFactory.keys())
        styleNames.sort()
        titles = [name for name in styleNames]
        styleNames = [name.lower() for name in styleNames]
        for i in range(len(titles)):
            if titles[i].lower() == pyzo.defaultQtStyleName.lower():
                titles[i] += " (%s)" % translate("menu", "default")
        self._qtThemeMenu.setOptions(titles, styleNames)
        self._qtThemeMenu.setCheckedOption(None,
pyzo.config.view.qtstyle.lower())
        # Build menu
        self.addItem(
            translate(
                "menu", "Select shell ::: Focus the cursor on the current
shell."
            ),
            icons.application_shell,
            self._selectShell,
        )
        self.addItem(
            translate(
                "menu", "Select editor ::: Focus the cursor on the current
editor."
            ),
            icons.application_edit,
            self._selectEditor,
        )
        self.addItem(
            translate(
                "menu", "Select previous file ::: Select the previously selected
file."
            ),
            icons.application_double,
            pyzo.editors._tabs.selectPreviousItem,
        )
        self.addSeparator()
        self.addEditorItem(
            translate("menu", "Show whitespace ::: Show spaces and tabs."),
            None,
            "showWhitespace",
            True,
        )
        self.addEditorItem(
            translate("menu", "Show line endings ::: Show the end of each
line."),
            None,
            "showLineEndings",
        )
        self.addEditorItem(
            translate(
                "menu",
                "Show indentation guides ::: Show vertical lines to indicate
indentation.",
            ),
            None,
            "showIndentationGuides",
        )
        self.addCheckItem(
            translate("menu", "Show status bar ::: Show status bar."),
            None,
            self._setStatusBar,
            None,
            pyzo.config.view.showStatusbar,
        )
        self.addSeparator()
        self.addEditorItem(
            translate(
                "menu",
                "Wrap long lines ::: Wrap lines that do not fit on the screen
(i.e. no horizontal scrolling).",
            ),
            None,
            "wrap",
        )
        self.addEditorItem(
            translate(
                "menu",
                "Highlight current line ::: Highlight the line where the cursor
is.",
            ),
            None,
            "highlightCurrentLine",
        )
        self.addEditorItem(
            translate(
                "menu",
                "Highlight brackets ::: Highlight matched and unmatched
brackets.",
            ),
            None,
            "highlightMatchingBracket",
            True,
        )
        self.addSeparator()
        self.addItem(
            translate("menu", "Previous cell ::: Go back to the previous 
cell."),
            None,
            self._previousCell,
        )
        self.addItem(
            translate("menu", "Next cell ::: Advance to the next cell."),
            None,
            self._nextCell,
        )
        self.addItem(
            translate(
                "menu",
                "Previous object ::: Go back to the previous top-level 
structure.",
            ),
            None,
            self._previousTopLevelObject,
        )
        self.addItem(
            translate(
                "menu", "Next object ::: Advance to the next top-level 
structure."
            ),
            None,
            self._nextTopLevelObject,
        )
        self.addSeparator()
        self.addMenu(self._edgeColumMenu, icons.text_padding_right)
        self.addMenu(FontMenu(self, translate("menu", "Font")), icons.style)
        self.addMenu(ZoomMenu(self, translate("menu", "Zooming")), 
icons.magnifier)
        self.addMenu(self._qtThemeMenu, icons.application_view_tile)
    def addEditorItem(self, name, icon, param, shellsToo=False):
        """
        Create a boolean item that reperesents a property of the editors,
        whose value is stored in pyzo.config.view.param
        """
        if hasattr(pyzo.config.view, param):
            default = getattr(pyzo.config.view, param)
        else:
            default = True
        self.addCheckItem(
            name,
            icon,
            lambda state, param: self._configEditor(state, param, shellsToo),
            param,
            default,
        )
    def _configEditor(self, state, param, shellsToo=False):
        """
        Callback for addEditorItem items
        """
        # Store this parameter in the config
        setattr(pyzo.config.view, param, state)
        # Apply to all editors, translate e.g. showWhitespace to
setShowWhitespace
        setter = "set" + param[0].upper() + param[1:]
        for editor in pyzo.editors:
            getattr(editor, setter)(state)
        if shellsToo:
            for shell in pyzo.shells:
                getattr(shell, setter)(state)
    def _selectShell(self):
        shell = pyzo.shells.getCurrentShell()
        if shell:
            shell.setFocus()
    def _selectEditor(self):
        editor = pyzo.editors.getCurrentEditor()
        if editor:
            editor.setFocus()
    def _setEdgeColumn(self, value):
        pyzo.config.view.edgeColumn = value
        for editor in pyzo.editors:
            editor.setLongLineIndicatorPosition(value)
    def _setQtTheme(self, value):
        pyzo.config.view.qtstyle = value
        pyzo.main.setQtStyle(value)
    def _setStatusBar(self, value):
        """
        Show or hide status bar.
        """
        pyzo.config.view.showStatusbar = value
        pyzo.main.statusBar().setVisible(value)
    def _previousCell(self):
        """
        Rewind the curser to the previous cell (starting with '##').
        """
        self._previousTopLevelObject(type="cell")
    def _nextCell(self):
        """
        Advance the curser to the next cell (starting with '##').
        """
        self._nextTopLevelObject(type="cell")
    def _previousTopLevelObject(self, type=None):
        # Get parser result
        result = pyzo.parser._getResult()
        if not result:
            return
        # Get editor
        editor = pyzo.editors.getCurrentEditor()
        if not editor:
            return
        # Get current line number
        ln = editor.textCursor().blockNumber()
        ln += 1  # is ln as in line number area
        runCursor = editor.textCursor()  # The part that should be run
        runCursor.movePosition(runCursor.StartOfBlock)
        # Find the object which starts above current curser
        # position if there is any and move there
        for object in reversed(result.rootItem.children):
            # If type given, only consider objects of that type
            if type and type != object.type:
                continue
            if ln and object.linenr < ln:
                startLineNr = object.linenr
                # Rewind cursor until the start of this object
                while True:
                    if not runCursor.block().previous().isValid():
                        return
                    runCursor.movePosition(runCursor.PreviousBlock)
                    if runCursor.blockNumber() == startLineNr - 1:
                        break
                cursor = editor.textCursor()
                cursor.setPosition(runCursor.position())
                editor.setTextCursor(cursor)
                return
    def _nextTopLevelObject(self, type=None):
        # Get parser result
        result = pyzo.parser._getResult()
        if not result:
            return
        # Get editor
        editor = pyzo.editors.getCurrentEditor()
        if not editor:
            return
        # Get current line number
        ln = editor.textCursor().blockNumber()
        ln += 1  # is ln as in line number area
        runCursor = editor.textCursor()  # The part that should be run
        runCursor.movePosition(runCursor.StartOfBlock)
        # Find the object which starts below current curser
        # position if there is any and move there
        for object in result.rootItem.children:
            # If type given, only consider objects of that type
            if type and type != object.type:
                continue
            if ln and object.linenr > ln:
                startLineNr = object.linenr
                endLineNr = object.linenr2
                # Advance cursor until the start of this object
                while True:
                    if not runCursor.block().next().isValid():
                        return
                    runCursor.movePosition(runCursor.NextBlock)
                    if runCursor.blockNumber() == startLineNr - 1:
                        break
                realCursorPosition = runCursor.position()
                # Advance cursor until the end of this object (to know
                # how far it extends and make sure it is most visible)
                while True:
                    if not runCursor.block().next().isValid():
                        break
                    runCursor.movePosition(runCursor.NextBlock)
                    if runCursor.blockNumber() == endLineNr - 1:
                        break
                cursor = editor.textCursor()
                cursor.setPosition(runCursor.position())
                editor.setTextCursor(cursor)
                cursor.setPosition(realCursorPosition)
                editor.setTextCursor(cursor)
                return
class ShellMenu(Menu):
    def __init__(self, parent=None, name="Shell"):
        self._shellCreateActions = []
        self._shellActions = []
        Menu.__init__(self, parent, name)
        pyzo.shells.currentShellChanged.connect(self.onCurrentShellChanged)
        self.aboutToShow.connect(self._updateShells)
    def onCurrentShellChanged(self):
        """Enable/disable shell actions based on wether a shell is available"""
        for shellAction in self._shellActions:
            shellAction.setEnabled(bool(pyzo.shells.getCurrentShell()))
    def buildShellActions(self):
        """Create the menu items which are also avaliable in the
        ShellTabContextMenu
        Returns a list of all items added"""
        icons = pyzo.icons
        return [
            self.addItem(
                translate("menu", "Clear screen ::: Clear the screen."),
                icons.application_eraser,
                self._shellAction,
                "clearScreen",
            ),
            self.addItem(
                translate(
                    "menu",
                    "Interrupt ::: Interrupt the current running code (does not
work for extension code).",
                ),
                icons.application_lightning,
                self._shellAction,
                "interrupt",
            ),
            self.addItem(
                translate("menu", "Restart ::: Terminate and restart the
interpreter."),
                icons.application_refresh,
                self._shellAction,
                "restart",
            ),
            # self.addItem(
            #     translate(
            #         "menu",
            #         "Terminate ::: Terminate the interpreter, leaving the 
shell open.",
            #     ),
            #     icons.application_delete,
            #     self._shellAction,
            #     "terminate",
            # ),
            self.addItem(
                translate(
                    "menu", "Close ::: Terminate the interpreter and close the 
shell."
                ),
                icons.cancel,
                self._shellAction,
                "closeShell",
            ),
        ]
    def buildShellDebugActions(self):
        """Create the menu items for debug shell actions.
        Returns a list of all items added"""
        icons = pyzo.icons
        return [
            self.addItem(
                translate("menu", "Debug next: proceed until next line"),
                icons.debug_next,
                self._debugAction,
                "NEXT",
            ),
            self.addItem(
                translate("menu", "Debug step into: proceed one step"),
                icons.debug_step,
                self._debugAction,
                "STEP",
            ),
            self.addItem(
                translate("menu", "Debug return: proceed until returns"),
                icons.debug_return,
                self._debugAction,
                "RETURN",
            ),
            self.addItem(
                translate("menu", "Debug continue: proceed to next breakpoint"),
                icons.debug_continue,
                self._debugAction,
                "CONTINUE",
            ),
            self.addItem(
                translate("menu", "Stop debugging"),
                icons.debug_quit,
                self._debugAction,
                "STOP",
            ),
        ]
    def getShell(self):
        """Returns the shell on which to apply the menu actions. Default is
        the current shell, this is overridden in the shell/shell tab context
        menus"""
        return pyzo.shells.getCurrentShell()
    def build(self):
        """Create the items for the shells menu"""
        # Normal shell actions
        self._shellActions = self.buildShellActions()
        self.addSeparator()
        # Debug stuff
        self._debug_clear_text = translate("menu", "Clear all {} breakpoints")
        self._debug_clear = self.addItem(
            "", pyzo.icons.bug_delete, self._clearBreakPoints
        )
        self._debug_pm = self.addItem(
            translate("menu", "Postmortem: debug from last traceback"),
            pyzo.icons.bug_delete,
            self._debugAction,
            "START",
        )
        self._shellDebugActions = self.buildShellDebugActions()
        #
        self.aboutToShow.connect(self._updateDebugButtons)
        self.addSeparator()
        # Shell config
        self.addItem(
            translate(
                "menu",
                "Edit shell configurations... ::: Add new shell configs and edit
interpreter properties.",
            ),
            pyzo.icons.application_wrench,
            self._editConfig2,
        )
        self.addItem(
            translate(
                "menu", "Create new Python environment... ::: Install 
miniconda."
            ),
            pyzo.icons.application_cascade,
            self._newPythonEnv,
        )
        self.addSeparator()
        # Add shell configs
        self._updateShells()
    def _updateShells(self):
        """Remove, then add the items for the creation of each shell"""
        for action in self._shellCreateActions:
            self.removeAction(action)
        self._shellCreateActions = []
        for i, config in enumerate(pyzo.config.shellConfigs2):
            name = translate("menu", "Create shell %s: (%s)") % (i + 1, 
config.name)
            action = self.addItem(
                name, pyzo.icons.application_add, pyzo.shells.addShell, config
            )
            self._shellCreateActions.append(action)
    def _updateDebugButtons(self):
        # Count breakpoints
        bpcount = 0
        for e in pyzo.editors:
            bpcount += len(e.breakPoints())
        self._debug_clear.setText(self._debug_clear_text.format(bpcount))
        # Determine state of PM and clear button
        debugmode = pyzo.shells._debugmode
        self._debug_pm.setEnabled(debugmode == 0)
        self._debug_clear.setEnabled(debugmode == 0)
        # The _shellDebugActions are enabled/disabled by the shellStack
    def _shellAction(self, action):
        """Call the method specified by 'action' on the current shell."""
        shell = self.getShell()
        if shell:
            # Call the specified action
            getattr(shell, action)()
    def _debugAction(self, action):
        shell = self.getShell()
        if shell:
            # Call the specified action
            command = action.upper()
            shell.executeCommand("DB %s\n" % command)
    def _clearBreakPoints(self, action=None):
        for e in pyzo.editors:
            e.clearBreakPoints()
    def _editConfig2(self):
        """Edit, add and remove configurations for the shells."""
        from pyzo.core.shellInfoDialog import ShellInfoDialog
        d = ShellInfoDialog()
        d.exec_()
    def _newPythonEnv(self):
        from pyzo.util.bootstrapconda import Installer
        d = Installer(pyzo.main)
        d.exec_()
class ShellButtonMenu(ShellMenu):
    def build(self):
        self._shellActions = []
        self.addItem(
            translate(
                "menu",
                "Edit shell configurations... ::: Add new shell configs and edit
interpreter properties.",
            ),
            pyzo.icons.application_wrench,
            self._editConfig2,
        )
        submenu = Menu(
            self,
            translate("menu", "New shell ... ::: Create new shell to run code
in."),
        )
        self._newShellMenu = self.addMenu(submenu, pyzo.icons.application_add)
        self.addSeparator()
    def _updateShells(self):
        """Remove, then add the items for the creation of each shell"""
        for action in self._shellCreateActions:
            self._newShellMenu.removeAction(action)
        self._shellCreateActions = []
        for i, config in enumerate(pyzo.config.shellConfigs2):
            name = translate("menu", "Create shell %s: (%s)") % (i + 1,
config.name)
            action = self._newShellMenu.addItem(
                name, pyzo.icons.application_add, pyzo.shells.addShell, config
            )
            self._shellCreateActions.append(action)
class ShellContextMenu(ShellMenu):
    """This is the context menu for the shell"""
    def __init__(self, shell, parent=None):
        ShellMenu.__init__(self, parent or shell, name="Shellcontextmenu")
        self._shell = shell
    def build(self):
        """Build menu"""
        icons = pyzo.icons
        self.addItem(
            translate(
                "menu",
                "Help on this expression ::: Show help for the selected
expression.",
            ),
            icons.help,
            self._editItemCallback,
            "helpOnText",
        )
        self.addSeparator()
        self.buildShellActions()
        # This is a subset of the edit menu. Copied manually.
        self.addSeparator()
        self.addItem(
            translate("menu", "Cut ::: Cut the selected text."),
            icons.cut,
            self._editItemCallback,
            "cut",
        )
        self.addItem(
            translate("menu", "Copy ::: Copy the selected text to the
clipboard."),
            icons.page_white_copy,
            self._editItemCallback,
            "copy",
        )
        self.addItem(
            translate("menu", "Paste ::: Paste the text that is now on the
clipboard."),
            icons.paste_plain,
            self._editItemCallback,
            "paste",
        )
        self.addItem(
            translate("menu", "Select all ::: Select all text."),
            icons.sum,
            self._editItemCallback,
            "selectAll",
        )
        self.addSeparator()
        self.addItem(
            translate("menu", "Open current directory in file browser"),
            None,
            self._editItemCallback,
            "opendir",
        )
        self.addItem(
            translate("menu", "Change current directory to the file browser's
path"),
            None,
            self._editItemCallback,
            "changedir",
        )
        self.addItem(
            translate("menu", "Change current directory to editor file path"),
            None,
            self._editItemCallback,
            "changedirtoeditor",
        )
    def getShell(self):
        """Shell actions of this menu operate on the shell specified in the
constructor"""
        return self._shell
    def _editItemCallback(self, action):
        # If the widget has a 'name' attribute, call it
        if action == "opendir":
            curdir = self._shell.get_kernel_cd()
            fileBrowser = pyzo.toolManager.getTool("pyzofilebrowser")
            if curdir and fileBrowser:
                fileBrowser.setPath(curdir)
        elif action == "changedir":
            fileBrowser = pyzo.toolManager.getTool("pyzofilebrowser")
            if fileBrowser:
                self._shell.executeCommand("cd " + fileBrowser.path() + "\n")
        elif action == "changedirtoeditor":
            msg = ""
            editor = pyzo.editors.getCurrentEditor()
            if editor is None:
                msg += translate("menu", "No editor selected.")
            # Show error dialog
            if msg:
                m = QtWidgets.QMessageBox(self)
                m.setWindowTitle(translate("menu dialog", "Could not change
dir"))
                m.setText(translate("menu", "Could not  change dir" + ":\n\n" +
msg))
                m.setIcon(m.Warning)
                m.exec_()
            else:
                self._shell.executeCommand(
                    "cd " + os.path.dirname(editor.filename) + "\n"
                )
        elif action == "helpOnText":
            self._shell.helpOnText(self._pos)
        else:
            getattr(self._shell, action)()
    def _updateShells(self):
        pass
class ShellTabContextMenu(ShellContextMenu):
    """The context menu for the shell tab is similar to the shell context menu,
    but only has the shell actions defined in ShellMenu.buildShellActions()"""
    def build(self):
        """Build menu"""
        self.buildShellActions()
    def _updateShells(self):
        pass
class EditorContextMenu(Menu):
    """This is the context menu for the editor"""
    def __init__(self, editor, name="EditorContextMenu"):
        self._editor = editor
        Menu.__init__(self, editor, name)
    def build(self):
        """Build menu"""
        icons = pyzo.icons
        self.addItem(
            translate(
                "menu",
                "Help on this expression ::: Show help for the selected
expression.",
            ),
            icons.help,
            self._editItemCallback,
            "helpOnText",
        )
        self.addSeparator()
        # This is a subset of the edit menu. Copied manually.
        self.addItem(
            translate("menu", "Cut ::: Cut the selected text."),
            icons.cut,
            self._editItemCallback,
            "cut",
        )
        self.addItem(
            translate("menu", "Copy ::: Copy the selected text to the
clipboard."),
            icons.page_white_copy,
            self._editItemCallback,
            "copy",
        )
        self.addItem(
            translate("menu", "Paste ::: Paste the text that is now on the
clipboard."),
            icons.paste_plain,
            self._editItemCallback,
            "paste",
        )
        self.addItem(
            translate("menu", "Select all ::: Select all text."),
            icons.sum,
            self._editItemCallback,
            "selectAll",
        )
        self.addSeparator()
        self.addItem(
            translate("menu", "Indent ::: Indent the selected line."),
            icons.text_indent,
            self._editItemCallback,
            "indentSelection",
        )
        self.addItem(
            translate("menu", "Dedent ::: Unindent the selected line."),
            icons.text_indent_remove,
            self._editItemCallback,
            "dedentSelection",
        )
        self.addItem(
            translate("menu", "Comment ::: Comment the selected line."),
            icons.comment_add,
            self._editItemCallback,
            "commentCode",
        )
        self.addItem(
            translate("menu", "Uncomment ::: Uncomment the selected line."),
            icons.comment_delete,
            self._editItemCallback,
            "uncommentCode",
        )
        self.addItem(
            translate(
                "menu", "Toggle Comment ::: Toggle comment for the selected
line."
            ),
            None,
            self._editItemCallback,
            "toggleCommentCode",
        )
        self.addItem(
            translate(
                "menu",
                "Justify comment/docstring::: Reshape the selected text so it is
aligned to around 70 characters.",
            ),
            icons.text_align_justify,
            self._editItemCallback,
            "justifyText",
        )
        self.addSeparator()
        self.addItem(
            translate(
                "menu", "Goto Definition ::: Go to definition of word under 
cursor."
            ),
            icons.debug_return,
            self._editItemCallback,
            "gotoDef",
        )
        self.addItem(
            translate("menu", "Open directory in file browser"),
            None,
            self._editItemCallback,
            "opendir",
        )
        self.addSeparator()
        self.addItem(
            translate(
                "menu",
                "Find or replace ::: Show find/replace widget. Initialize with 
selected text.",
            ),
            icons.find,
            pyzo.editors._findReplace.startFind,
        )
        self.addItem(
            translate(
                "menu",
                "Find selection ::: Find the next occurrence of the selected 
text.",
            ),
            None,
            pyzo.editors._findReplace.findSelection,
        )
        self.addItem(
            translate(
                "menu",
                "Find selection backward ::: Find the previous occurrence of the 
selected text.",
            ),
            None,
            pyzo.editors._findReplace.findSelectionBw,
        )
        # This is a subset of the run menu. Copied manually.
        self.addSeparator()
        self.addItem(
            translate(
                "menu",
                "Run selection ::: Run the current editor's selected lines,
selected words on the current line, or current line if there is no selection.",
            ),  # noqa
            icons.run_lines,
            self._runSelected,
        )
    def _editItemCallback(self, action):
        # If the widget has a 'name' attribute, call it
        if action == "opendir":
            fileBrowser = pyzo.toolManager.getTool("pyzofilebrowser")
            if fileBrowser:
                fileBrowser.setPath(os.path.dirname(self._editor.filename))
        elif action == "helpOnText":
            self._editor.helpOnText(self._pos)
        else:
            getattr(self._editor, action)()
    def _runSelected(self):
        runMenu = pyzo.main.menuBar()._menumap["run"]
        runMenu._runSelected()
class EditorTabContextMenu(Menu):
    def __init__(self, *args, **kwds):
        Menu.__init__(self, *args, **kwds)
        self._index = -1
    def setIndex(self, index):
        self._index = index
    def build(self):
        """Build menu"""
        icons = pyzo.icons
        # Copied (and edited) manually from the File memu
        self.addItem(
            translate("menu", "Save ::: Save the current file to disk."),
            icons.disk,
            self._fileAction,
            "saveFile",
        )
        self.addItem(
            translate(
                "menu", "Save as... ::: Save the current file under another
name."
            ),
            icons.disk_as,
            self._fileAction,
            "saveFileAs",
        )
        self.addItem(
            translate("menu", "Close ::: Close the current file."),
            icons.page_delete,
            self._fileAction,
            "closeFile",
        )
        self.addItem(
            translate("menu", "Close others::: Close all files but this one."),
            None,
            self._fileAction,
            "close_others",
        )
        self.addItem(
            translate("menu", "Close all ::: Close all files."),
            icons.page_delete_all,
            self._fileAction,
            "close_all",
        )
        self.addItem(
            translate("menu", "Rename ::: Rename this file."),
            None,
            self._fileAction,
            "rename",
        )
        self.addSeparator()
        self.addItem(
            translate("menu", "Copy path ::: Copy the full path of this file."),
            None,
            self._fileAction,
            "copypath",
        )
        self.addItem(
            translate("menu", "Open directory in file browser"),
            None,
            self._fileAction,
            "opendir",
        )
        self.addSeparator()
        # todo: remove feature to pin files?
        self.addItem(
            translate("menu", "Pin/Unpin ::: Pinned files get closed less 
easily."),
            None,
            self._fileAction,
            "pin",
        )
        self.addItem(
            translate(
                "menu",
                "Set/Unset as MAIN file ::: The main file can be run while 
another file is selected.",
            ),
            icons.star,
            self._fileAction,
            "main",
        )
        self.addSeparator()
        self.addItem(
            translate("menu", "Run file ::: Run the code in this file."),
            icons.run_file,
            self._fileAction,
            "run",
        )
        self.addItem(
            translate(
                "menu",
                "Run file as script ::: Run this file as a script (restarts the 
interpreter).",
            ),
            icons.run_file_script,
            self._fileAction,
            "run_script",
        )
    def _fileAction(self, action):
        """Call the method specified by 'action' on the selected shell"""
        item = pyzo.editors._tabs.getItemAt(self._index)
        if action in ["saveFile", "saveFileAs", "closeFile"]:
            getattr(pyzo.editors, action)(item.editor)
        elif action == "close_others" or action == "close_all":
            if action == "close_all":
                item = None  # The item not to be closed is not there
            items = pyzo.editors._tabs.items()
            for i in reversed(range(pyzo.editors._tabs.count())):
                if items[i] is item or items[i].pinned:
                    continue
                pyzo.editors._tabs.tabCloseRequested.emit(i)
        elif action == "rename":
            filename = item.filename
            pyzo.editors.saveFileAs(item.editor)
            if item.filename != filename:
                try:
                    os.remove(filename)
                except Exception:
                    pass
        elif action == "copypath":
            filename = item.filename
            QtWidgets.qApp.clipboard().setText(filename)
        elif action == "opendir":
            fileBrowser = pyzo.toolManager.getTool("pyzofilebrowser")
            if fileBrowser:
                fileBrowser.setPath(os.path.dirname(item.filename))
        elif action == "pin":
            item._pinned = not item._pinned
        elif action == "main":
            if pyzo.editors._tabs._mainFile == item.id:
                pyzo.editors._tabs._mainFile = None
            else:
                pyzo.editors._tabs._mainFile = item.id
        elif action == "run":
            menu = pyzo.main.menuBar().findChild(RunMenu)
            if menu:
                menu._runFile((False, False), item.editor)
        elif action == "run_script":
            menu = pyzo.main.menuBar().findChild(RunMenu)
            if menu:
                menu._runFile((True, False), item.editor)
        pyzo.editors._tabs.updateItems()
class RunMenu(Menu):
    def build(self):
        icons = pyzo.icons
        self.addItem(
            translate(
                "menu",
                "Run file as script ::: Restart and run the current file as a
script.",
            ),  # noqa
            icons.run_file_script,
            self._runFile,
            (True, False),
        )
        self.addItem(
            translate(
                "menu",
                "Run main file as script ::: Restart and run the main file as a
script.",
            ),  # noqa
            icons.run_mainfile_script,
            self._runFile,
            (True, True),
        )
        self.addSeparator()
        self.addItem(
            translate(
                "menu",
                "Execute selection ::: Execute the current editor's selected
lines, selected words on the current line, or current line if there is no
selection.",
            ),  # noqa
            icons.run_lines,
            self._runSelected,
        )
        self.addItem(
            translate(
                "menu",
                "Execute cell ::: Execute the current editors's cell in the
current shell.",
            ),  # noqa
            icons.run_cell,
            self._runCell,
        )
        # In the _runFile calls, the parameter specifies (asScript, mainFile)
        self.addItem(
            translate(
                "menu",
                "Execute file ::: Execute the current file in the current
shell.",
            ),
            icons.run_file,
            self._runFile,
            (False, False),
        )
        self.addItem(
            translate(
                "menu",
                "Execute main file ::: Execute the main file in the current
shell.",
            ),
            icons.run_mainfile,
            self._runFile,
            (False, True),
        )
        self.addSeparator()
        self.addItem(
            translate("menu", "Execute selection and advance"),
            icons.run_lines,
            self._runSelectedAdvance,
        )
        self.addItem(
            translate(
                "menu",
                "Execute cell and advance ::: Execute the current editors's cell
and advance to the next cell.",
            ),
            icons.run_cell,
            self._runCellAdvance,
        )
        self.addSeparator()
        self.addCheckItem(
            translate(
                "menu",
                "Change directory when executing file ::: like Run File As
Script does",
            ),
            None,
            self._cdonfileexec,
            None,
            pyzo.config.settings.changeDirOnFileExec,
        )
        self.addItem(
            translate(
                "menu",
                "Help on running code ::: Open the pyzo wizard at the page about
running code.",
            ),
            icons.information,
            self._showHelp,
        )
    def _cdonfileexec(self, value):
        pyzo.config.settings.changeDirOnFileExec = bool(value)
    def _showHelp(self):
        """Show more information about ways to run code."""
        from pyzo.util.pyzowizard import PyzoWizard
        w = PyzoWizard(self)
        w.show("RuncodeWizardPage1")  # Start wizard at page about running code
    def _getShellAndEditor(self, what, mainEditor=False):
        """Get the shell and editor. Shows a warning dialog when one of
        these is not available.
        """
        # Init empty error message
        msg = ""
        # Get shell
        shell = pyzo.shells.getCurrentShell()
        if shell is None:
            msg += translate("menu", "No shell to run code in.").rstrip() + " "
            # shell = pyzo.shells.addShell()  # issue #335, does not work,
somehow
        # Get editor
        if mainEditor:
            editor = pyzo.editors.getMainEditor()
            if editor is None:
                msg += translate("menu", "There is no main file selected.")
        else:
            editor = pyzo.editors.getCurrentEditor()
            if editor is None:
                msg += translate("menu", "No editor selected.")
        # Show error dialog
        if msg:
            m = QtWidgets.QMessageBox(self)
            m.setWindowTitle(translate("menu dialog", "Could not run"))
            m.setText(translate("menu", "Could not run " + what + ":\n\n" +
msg))
            m.setIcon(m.Warning)
            m.exec_()
        # Return
        return shell, editor
    def _advance(self, runCursor):
        # Get editor and shell
        shell, editor = self._getShellAndEditor("selection")
        if not shell or not editor:
            return
        cursor = editor.textCursor()
        cursor.setPosition(runCursor.position())
        cursor.movePosition(cursor.NextBlock)
        editor.setTextCursor(cursor)
    def _runSelectedAdvance(self):
        self._runSelected(advance=True)
    def _runSelected(self, advance=False):
        """Run the selected whole lines in the current shell."""
        # Get editor and shell
        shell, editor = self._getShellAndEditor("selection")
        if not shell or not editor:
            return
        # Get position to sample between (only sample whole lines)
        screenCursor = editor.textCursor()  # Current selection in the editor
        runCursor = editor.textCursor()  # The part that should be run
        runCursor.setPosition(screenCursor.selectionStart())
        runCursor.movePosition(runCursor.StartOfBlock)  # This also moves the
anchor
        lineNumber1 = runCursor.blockNumber()
        runCursor.setPosition(screenCursor.selectionEnd(), runCursor.KeepAnchor)
        if not (screenCursor.hasSelection() and runCursor.atBlockStart()):
            # If the end of the selection is at the beginning of a block, don't
extend it
            runCursor.movePosition(runCursor.EndOfBlock, runCursor.KeepAnchor)
        lineNumber2 = runCursor.blockNumber()
        # Does this look like a statement?
        isStatement = lineNumber1 == lineNumber2 and screenCursor.hasSelection()
        if isStatement:
            # Get source code of statement
            code = screenCursor.selectedText().replace("\u2029", "\n").strip()
            # add code to history
            pyzo.command_history.append(code)
            # Execute statement
            shell.executeCommand(code + "\n")
        else:
            # Get source code
            code = runCursor.selectedText().replace("\u2029", "\n")
            # Notify user of what we execute
            self._showWhatToExecute(editor, runCursor)
            # Get filename and run code
            fname = editor.id()  # editor._name or editor._filename
            shell.executeCode(code, fname, lineNumber1)
        if advance:
            self._advance(runCursor)
    def _runCellAdvance(self):
        self._runCell(advance=True)
    def _runCell(self, advance=False):
        """Run the code between two cell separaters ('##')."""
        # TODO: ignore ## in multi-line strings
        # Maybe using source-structure information?
        # Get editor and shell
        shell, editor = self._getShellAndEditor("cell")
        if not shell or not editor:
            return
        cellName = ""
        # Get current cell
        # Move up until the start of document
        # or right after a line starting with '##'
        runCursor = editor.textCursor()  # The part that should be run
        runCursor.movePosition(runCursor.StartOfBlock)
        while True:
            line = runCursor.block().text().lstrip()
            if (
                line.startswith("##")
                or line.startswith("#%%")
                or line.startswith("# %%")
            ):
                # ## line, move to the line following this one
                if not runCursor.block().next().isValid():
                    # The user tried to execute the last line of a file which
                    # started with ##. Do nothing
                    return
                runCursor.movePosition(runCursor.NextBlock)
                cellName = line.lstrip("#% ").strip()
                break
            if not runCursor.block().previous().isValid():
                break  # Start of document
            runCursor.movePosition(runCursor.PreviousBlock)
        # This is the line number of the start
        lineNumber = runCursor.blockNumber()
        if len(cellName) > 20:
            cellName = cellName[:17] + "..."
        # Move down until a line before one starting with'##'
        # or to end of document
        while True:
            line = runCursor.block().text().lstrip()
            if (
                line.startswith("##")
                or line.startswith("#%%")
                or line.startswith("# %%")
            ):
                # This line starts with ##, move to the end of the previous one
                runCursor.movePosition(runCursor.Left, runCursor.KeepAnchor)
                break
            if not runCursor.block().next().isValid():
                # Last block of the document, move to the end of the line
                runCursor.movePosition(runCursor.EndOfBlock, 
runCursor.KeepAnchor)
                break
            runCursor.movePosition(runCursor.NextBlock, runCursor.KeepAnchor)
        # Get source code
        code = runCursor.selectedText().replace("\u2029", "\n")
        # Notify user of what we execute
        self._showWhatToExecute(editor, runCursor)
        # Get filename and run code
        fname = editor.id()  # editor._name or editor._filename
        shell.executeCode(
            code, fname, lineNumber, cellName or " "
        )  # ensure that bool(cellName) == True
        if advance:
            self._advance(runCursor)
    def _showWhatToExecute(self, editor, runCursor=None):
        # Get runCursor for whole document if not given
        if runCursor is None:
            runCursor = editor.textCursor()
            runCursor.movePosition(runCursor.Start)
            runCursor.movePosition(runCursor.End, runCursor.KeepAnchor)
        editor.showRunCursor(runCursor)
    def _getCodeOfFile(self, editor):
        # Obtain source code
        text = editor.toPlainText()
        # Show what we execute
        self._showWhatToExecute(editor)
        # Get filename and return
        fname = editor.id()  # editor._name or editor._filename
        return fname, text
    def _runFile(self, runMode, givenEditor=None):
        """Run a file
        runMode is a tuple (asScript, mainFile)
        """
        asScript, mainFile = runMode
        # Get editor and shell
        description = "main file" if mainFile else "file"
        if asScript:
            description += translate("menu", " (as script)")
        shell, editor = self._getShellAndEditor(description, mainFile)
        if givenEditor:
            editor = givenEditor
        if not shell or not editor:
            return
        if asScript:
            # Go
            self._runScript(editor, shell)
        else:
            # Obtain source code and fname
            fname, text = self._getCodeOfFile(editor)
            shell.executeCode(
                text, fname, changeDir=pyzo.config.settings.changeDirOnFileExec
            )
    def _runScript(self, editor, shell):
        # Obtain fname and try running
        err = ""
        if editor._filename:
            saveOk = pyzo.editors.saveFile(editor)  # Always try to save
            if saveOk or not editor.document().isModified():
                self._showWhatToExecute(editor)
                if shell._startup_info.get("ipython", "") == "yes":
                    # If we have a ipython shell we use %run -i instead
                    # This works better when python autoreload is used
                    d = os.path.normpath(
                        os.path.normcase(os.path.dirname(editor._filename))
                    )
                    shell._ctrl_command.send('%%cd "%s"\n' % d)
                    shell._ctrl_command.send('%%run -i "%s"\n' %
editor._filename)
                else:
                    shell.restart(editor._filename)
            else:
                err = translate("menu", "Could not save the file.")
        else:
            err = translate("menu", "Can only run scripts that are in the file 
system.")
        # If not success, notify
        if err:
            m = QtWidgets.QMessageBox(self)
            m.setWindowTitle(translate("menu dialog", "Could not run script."))
            m.setText(err)
            m.setIcon(m.Warning)
            m.exec_()
class ToolsMenu(Menu):
    def __init__(self, *args, **kwds):
        self._toolActions = []
        Menu.__init__(self, *args, **kwds)
    def build(self):
        self.addItem(
            translate("menu", "Reload tools ::: For people who develop tools."),
            pyzo.icons.plugin_refresh,
            pyzo.toolManager.reloadTools,
        )
        self.addSeparator()
        self.onToolInstanceChange()  # Build initial menu
        pyzo.toolManager.toolInstanceChange.connect(self.onToolInstanceChange)
    def onToolInstanceChange(self):
        # Remove all exisiting tools from the menu
        for toolAction in self._toolActions:
            self.removeAction(toolAction)
        # Add all tools, with checkmarks for those that are active
        self._toolActions = []
        for tool in pyzo.toolManager.getToolInfo():
            action = self.addCheckItem(
                tool.name,
                pyzo.icons.plugin,
                tool.menuLauncher,
                selected=bool(tool.instance),
            )
            self._toolActions.append(action)
class HelpMenu(Menu):
    def build(self):
        icons = pyzo.icons
        self.addUrlItem(
            translate(
                "menu", "Pyzo website ::: Open the Pyzo website in your
browser."
            ),
            icons.help,
            "http://pyzo.org",
        )
        self.addUrlItem(
            translate("menu", "Pyzo guide ::: Open the Pyzo guide in your
browser."),
            icons.help,
            "http://guide.pyzo.org",
        )
        self.addItem(
            translate("menu", "Pyzo wizard ::: Get started quickly."),
            icons.wand,
            self._showPyzoWizard,
        )
        self.addSeparator()
        self.addUrlItem(
            translate("menu", "Ask a question ::: Need help?"),
            icons.comments,
            "http://community.pyzo.org",
        )
        self.addUrlItem(
            translate(
                "menu",
                "Report an issue ::: Did you found a bug in Pyzo, or do you have
a feature request?",
            ),
            icons.error_add,
            "http://issues.pyzo.org",
        )
        self.addItem(
            translate(
                "menu",
                "Local documentation ::: Documentation on Python and the Scipy
Stack.",
            ),
            icons.help,
            self._showPyzoDocs,
        )
        self.addSeparator()
        # self.addItem(translate("menu", "View code license ::: Legal stuff."),
        #    icons.script, lambda:
pyzo.editors.loadFile(os.path.join(pyzo.pyzoDir,"license.txt")))
        self.addItem(
            translate(
                "menu", "Check for updates ::: Are you using the latest
version?"
            ),
            icons.application_go,
            self._checkUpdates,
        )
        self.addItem(
            translate("menu", "About Pyzo ::: More information about Pyzo."),
            icons.information,
            self._aboutPyzo,
        )
    def addUrlItem(self, name, icon, url):
        self.addItem(name, icon, lambda: webbrowser.open(url))
    def _showPyzoWizard(self):
        from pyzo.util.pyzowizard import PyzoWizard
        w = PyzoWizard(self)
        w.show()  # Use show() instead of exec_() so the user can interact with
pyzo
    def _checkUpdates(self):
        """Check whether a newer version of pyzo is available."""
        # Get versions available
        url = "https://api.github.com/repos/pyzo/pyzo/releases"
        releases = json.loads(urlopen(url).read())
        versions = []
        for release in releases:
            tag = release.get("tag_name", "")
            if tag.startswith("v"):
                version = tuple(int(i) for i in tag[1:].split("."))
                versions.append(version)
        versions.sort()
        latest_version = ".".join(str(i) for i in versions[-1]) if versions else
"?"
        # Define message
        text = "Your version of Pyzo is: {}\n"
        text += "Latest available version is: {}\n\n"
        text = text.format(pyzo.__version__, latest_version)
        text += "Do you want to open the download page?\n"
        # Show message box
        m = QtWidgets.QMessageBox(self)
        m.setWindowTitle(translate("menu dialog", "Check for the latest 
version."))
        m.setStandardButtons(m.Yes | m.Cancel)
        m.setDefaultButton(m.Cancel)
        m.setText(text)
        m.setIcon(m.Information)
        result = m.exec_()
        # Goto webpage if user chose to
        if result == m.Yes:
            webbrowser.open("http://pyzo.org/start.html")
    def _aboutPyzo(self):
        from pyzo.core.about import AboutDialog
        m = AboutDialog(self)
        m.exec_()
    def _showPyzoDocs(self):
        # Show widget with docs:
        self._assistant = PyzoAssistant()
        self._assistant.show()
class AutocompMenu(Menu):
    """
    Menu for the user to control autocompletion.
    """
    def build(self):
        # Part for selecting mode
        modes = [
            translate("menu", "No autocompletion"),
            translate("menu", "Automatic popup"),
            translate("menu", "Only show popup when pressing Tab"),
        ]
        for value, mode in enumerate(modes):
            self.addGroupItem(mode, None, self._setMode, value, group="mode")
        self.setCheckedOption("mode", pyzo.config.settings.autoComplete)
        self.addSeparator()
        # Part for accept key
        accept_keys = [
            ("Tab", translate("menu", "Tab")),
            ("Enter", translate("menu", "Enter")),
            ("Tab, Enter", translate("menu", "Tab, Enter")),
            ("Tab, (, [", translate("menu", "Tab, (, [")),
            ("Tab, Enter, (, [", translate("menu", "Tab, Enter, (, [")),
        ]
        prefix = translate("menu", "Accept autocompletion with:")
        for keys, display in accept_keys:
            self.addGroupItem(
                prefix + " " + display,
                None,
                self._setAcceptKeys,
                keys,
                group="acceptkeys",
            )
        self.setCheckedOption(
            "acceptkeys", pyzo.config.settings.autoComplete_acceptKeys
        )
        self.addSeparator()
        # Booleans
        self.addCheckItem(
            translate(
                "menu",
                "Autocomplete keywords ::: The autocompletion list includes
keywords.",
            ),
            None,
            self._setCompleteKeywords,
            None,
            pyzo.config.settings.autoComplete_keywords,
        )
        self.addSeparator()
        # auto closing options
        self.addCheckItem(
            translate(
                "menu", "Auto close quotes ::: Auto close single and double
quotes."
            ),
            None,
            self._setQuotes,
            None,
            pyzo.config.settings.autoClose_Quotes,
        )
        self.addCheckItem(
            translate("menu", "Auto close brackets ::: Auto close ( { [ ] }
)."),
            None,
            self._setBrackets,
            None,
            pyzo.config.settings.autoClose_Brackets,
        )
    def _setAcceptKeys(self, autocompkeys):
        # Skip if setting is not changes
        if pyzo.config.settings.autoComplete_acceptKeys == autocompkeys:
            return
        # Save new setting
        pyzo.config.settings.autoComplete_acceptKeys = autocompkeys
        # Apply
        for e in pyzo.editors:
            e.setAutoCompletionAcceptKeysFromStr(autocompkeys)
        for s in pyzo.shells:
            s.setAutoCompletionAcceptKeysFromStr(autocompkeys)
    def _setMode(self, autocompmode):
        pyzo.config.settings.autoComplete = int(autocompmode)
    def _setCompleteKeywords(self, value):
        pyzo.config.settings.autoComplete_keywords = bool(value)
    def _setQuotes(self, value):
        # Set automatic insertion of single and double quotes
        pyzo.config.settings.autoClose_Quotes = bool(value)
    def _setBrackets(self, value):
        # Set automatic insertion of parenthesis, braces and brackets
        pyzo.config.settings.autoClose_Brackets = bool(value)
class SettingsMenu(Menu):
    def build(self):
        icons = pyzo.icons
        # Create language menu
        from pyzo.util._locale import LANGUAGES, LANGUAGE_SYNONYMS
        # Update language setting if necessary
        cur = pyzo.config.settings.language
        pyzo.config.settings.language = LANGUAGE_SYNONYMS.get(cur, cur)
        # Create language menu
        t = translate("menu", "Select language ::: The language used by Pyzo.")
        self._languageMenu = GeneralOptionsMenu(self, t, self._selectLanguage)
        values = [key for key in sorted(LANGUAGES)]
        self._languageMenu.setOptions(values, values)
        self._languageMenu.setCheckedOption(None, pyzo.config.settings.language)
        self.addBoolSetting(
            translate(
                "menu",
                "Automatically indent ::: Indent when pressing enter after a
colon.",
            ),
            "autoIndent",
            lambda state, key: [e.setAutoIndent(state) for e in pyzo.editors],
        )
        self.addBoolSetting(
            translate(
                "menu", "Enable calltips ::: Show calltips with function 
signatures."
            ),
            "autoCallTip",
        )
        self.addMenu(AutocompMenu(self, translate("menu", "Autocompletion")))
        self.addSeparator()
        self.addItem(
            translate(
                "menu", "Edit key mappings... ::: Edit the shortcuts for menu 
items."
            ),
            icons.keyboard,
            lambda: KeymappingDialog().exec_(),
        )
        self.addItem(
            translate(
                "menu", "Edit syntax styles... ::: Change the coloring of your 
code."
            ),
            icons.style,
            lambda: EditColorDialog().exec_(),
        )
        self.addMenu(self._languageMenu, icons.flag_green)
        self.addItem(
            translate("menu", "Advanced settings... ::: Configure Pyzo even 
further."),
            icons.cog,
            lambda: AdvancedSettings().exec_(),
        )
    def addBoolSetting(self, name, key, callback=None):
        def _callback(state, key):
            setattr(pyzo.config.settings, key, state)
            if callback is not None:
                callback(state, key)
        self.addCheckItem(
            name, None, _callback, key, getattr(pyzo.config.settings, key)
        )  # Default value
    def _selectLanguage(self, languageName):
        # Skip if the same
        if pyzo.config.settings.language == languageName:
            return
        # Save new language
        pyzo.config.settings.language = languageName
        # Notify user
        text = translate(
            "menu dialog",
            """
        The language has been changed.
        Pyzo needs to restart for the change to take effect.
        """,
        )
        m = QtWidgets.QMessageBox(self)
        m.setWindowTitle(translate("menu dialog", "Language changed"))
        m.setText(unwrapText(text))
        m.setIcon(m.Information)
        m.exec_()
## Classes to enable editing the key mappings
class KeyMapModel(QtCore.QAbstractItemModel):
    """The model to view the structure of the menu and the shortcuts
    currently mapped."""
    def __init__(self, *args):
        QtCore.QAbstractItemModel.__init__(self, *args)
        self._root = None
    def setRootMenu(self, menu):
        """Call this after starting."""
        self._root = menu
    def data(self, index, role):
        if not index.isValid() or role not in [0, 8]:
            return None
        # get menu or action item
        item = index.internalPointer()
        # get text and shortcuts
        key1, key2 = "", ""
        if isinstance(item, QtWidgets.QMenu):
            value = item.title()
        else:
            value = item.text()
            if not value:
                value = "-" * 10
            elif index.column() > 0:
                key1, key2 = " ", " "
                shortcuts = getShortcut(item)
                if shortcuts[0]:
                    key1 = shortcuts[0]
                if shortcuts[1]:
                    key2 = shortcuts[1]
        # translate to text for the user
        key1 = translateShortcutToOSNames(key1)
        key2 = translateShortcutToOSNames(key2)
        # obtain value
        value = [value, key1, key2, ""][index.column()]
        # return
        if role == 0:
            # display role
            return value
        elif role == 8:
            # 8: BackgroundRole
            if not value:
                return None
            elif index.column() == 1:
                return QtGui.QBrush(QtGui.QColor(200, 220, 240))
            elif index.column() == 2:
                return QtGui.QBrush(QtGui.QColor(210, 230, 250))
            else:
                return None
        else:
            return None
    def rowCount(self, parent):
        if parent.isValid():
            menu = parent.internalPointer()
            return len(menu.actions())
        else:
            return len(self._root.actions())
    def columnCount(self, parent):
        return 4
    def headerData(self, section, orientation, role):
        if role == 0:  # and orientation==1:
            tmp = ["Menu action", "Shortcut 1", "Shortcut 2", ""]
            return tmp[section]
    def parent(self, index):
        if not index.isValid():
            return QtCore.QModelIndex()
        item = index.internalPointer()
        pitem = item.parent()
        if pitem is self._root:
            return QtCore.QModelIndex()
        else:
            if pitem is None:
                # menu seems to be hidden for key mapping settings dialog
                return QtCore.QModelIndex()
            L = pitem.parent().actions()
            row = 0
            if pitem in L:
                row = L.index(pitem)
            return self.createIndex(row, 0, pitem)
    def hasChildren(self, index):
        # no items have parents (except the root item)
        if index.row() < 0:
            return True
        else:
            return isinstance(index.internalPointer(), QtWidgets.QMenu)
    def index(self, row, column, parent):
        if not self.hasIndex(row, column, parent):
            return QtCore.QModelIndex()
        # establish parent
        if not parent.isValid():
            parentMenu = self._root
        else:
            parentMenu = parent.internalPointer()
        # produce index and make menu if the action represents a menu
        childAction = parentMenu.actions()[row]
        childMenu = childAction.parent()
        if childMenu is not parentMenu and childMenu.actions():
            childAction = childMenu
        if isinstance(childMenu, Menu) and getattr(childMenu,
'_hide_in_key_map_dialog', False):
            childAction = Menu._dummyActionForHiddenEntryInKeyMapDialog
        return self.createIndex(row, column, childAction)
        # This is the trick. The internal pointer is the way to establish
        # correspondence between ModelIndex and underlying data.
# Key to string mappings
k = QtCore.Qt
keymap = {
    k.Key_Enter: "Enter",
    k.Key_Return: "Return",
    k.Key_Escape: "Escape",
    k.Key_Tab: "Tab",
    k.Key_Backspace: "Backspace",
    k.Key_Pause: "Pause",
    k.Key_Backtab: "Tab",  # Backtab is actually shift+tab
    k.Key_F1: "F1",
    k.Key_F2: "F2",
    k.Key_F3: "F3",
    k.Key_F4: "F4",
    k.Key_F5: "F5",
    k.Key_F6: "F6",
    k.Key_F7: "F7",
    k.Key_F8: "F8",
    k.Key_F9: "F9",
    k.Key_F10: "F10",
    k.Key_F11: "F11",
    k.Key_F12: "F12",
    k.Key_Space: "Space",
    k.Key_Delete: "Delete",
    k.Key_Insert: "Insert",
    k.Key_Home: "Home",
    k.Key_End: "End",
    k.Key_PageUp: "PageUp",
    k.Key_PageDown: "PageDown",
    k.Key_Left: "Left",
    k.Key_Up: "Up",
    k.Key_Right: "Right",
    k.Key_Down: "Down",
}
class KeyMapLineEdit(QtWidgets.QLineEdit):
    """A modified version of a lineEdit object that catches the key event
    and displays "Ctrl" when control was pressed, and similarly for alt and
    shift, function keys and other keys.
    """
    textUpdate = QtCore.Signal()
    def __init__(self, *args, **kwargs):
        QtWidgets.QLineEdit.__init__(self, *args, **kwargs)
        self.clear()
        # keep a list of native keys, so that we can capture for example
        # "shift+]". If we would use text(), we can only capture "shift+}"
        # which is not a valid shortcut.
        self._nativeKeys = {}
    # Override setText, text and clear, so as to be able to set shortcuts like
    # Ctrl+A, while the actually displayed value is an OS shortcut (e.g. on Mac
    # Cmd-symbol + A)
    def setText(self, text):
        QtWidgets.QLineEdit.setText(self, translateShortcutToOSNames(text))
        self._shortcut = text
    def text(self):
        return self._shortcut
    def clear(self):
        QtWidgets.QLineEdit.setText(self, "<enter key combination here>")
        self._shortcut = ""
    def focusInEvent(self, event):
        # self.clear()
        QtWidgets.QLineEdit.focusInEvent(self, event)
    def event(self, event):
        # Override event handler to enable catching the Tab key
        # If the event is a KeyPress or KeyRelease, handle it with
        # self.keyPressEvent or keyReleaseEvent
        if event.type() == event.KeyPress:
            self.keyPressEvent(event)
            return True  # Mark as handled
        if event.type() == event.KeyRelease:
            self.keyReleaseEvent(event)
            return True  # Mark as handled
        # Default: handle events as usual
        return QtWidgets.QLineEdit.event(self, event)
    def keyPressEvent(self, event):
        # get key codes
        key = event.key()
        nativekey = event.nativeVirtualKey()
        # try to get text
        if nativekey < 128 and sys.platform != "darwin":
            text = chr(nativekey).upper()
        elif key < 128:
            text = chr(key).upper()
        else:
            text = ""
        # do we know this specic key or this native key?
        if key in keymap:
            text = keymap[key]
        elif nativekey in self._nativeKeys:
            text = self._nativeKeys[nativekey]
        # apply!
        if text:
            storeNativeKey, text0 = True, text
            if QtWidgets.qApp.keyboardModifiers() & k.AltModifier:
                text = "Alt+" + text
            if QtWidgets.qApp.keyboardModifiers() & k.ShiftModifier:
                text = "Shift+" + text
                storeNativeKey = False
            if QtWidgets.qApp.keyboardModifiers() & k.ControlModifier:
                text = "Ctrl+" + text
            if QtWidgets.qApp.keyboardModifiers() & k.MetaModifier:
                text = "Meta+" + text
            self.setText(text)
            if storeNativeKey and nativekey:
                # store native key if shift was not pressed.
                self._nativeKeys[nativekey] = text0
        # notify listeners
        self.textUpdate.emit()
class KeyMapEditDialog(QtWidgets.QDialog):
    """The prompt that is shown when double clicking
    a keymap in the tree.
    It notifies the user when the entered shortcut is already used
    elsewhere and applies the shortcut (removing it elsewhere if
    required) when the apply button is pressed.
    """
    def __init__(self, *args):
        QtWidgets.QDialog.__init__(self, *args)
        # set title
        self.setWindowTitle(translate("menu dialog", "Edit shortcut mapping"))
        # set size
        size = 400, 140
        offset = 5
        size2 = size[0], size[1] + offset
        self.resize(*size2)
        self.setMaximumSize(*size2)
        self.setMinimumSize(*size2)
        self._label = QtWidgets.QLabel("", self)
        self._label.setAlignment(QtCore.Qt.AlignTop | QtCore.Qt.AlignLeft)
        self._label.resize(size[0] - 20, 100)
        self._label.move(10, 2)
        self._line = KeyMapLineEdit("", self)
        self._line.resize(size[0] - 80, 20)
        self._line.move(10, 90)
        self._clear = QtWidgets.QPushButton("Clear", self)
        self._clear.resize(50, 20)
        self._clear.move(size[0] - 60, 90)
        self._apply = QtWidgets.QPushButton("Apply", self)
        self._apply.resize(50, 20)
        self._apply.move(size[0] - 120, 120)
        self._cancel = QtWidgets.QPushButton("Cancel", self)
        self._cancel.resize(50, 20)
        self._cancel.move(size[0] - 60, 120)
        # callbacks
        self._line.textUpdate.connect(self.onEdit)
        self._clear.clicked.connect(self.onClear)
        self._apply.clicked.connect(self.onAccept)
        self._cancel.clicked.connect(self.close)
        # stuff to fill in later
        self._fullname = ""
        self._intro = ""
        self._isprimary = True
    def setFullName(self, fullname, isprimary):
        """To be called right after initialization to let the user
        know what he's updating, and show the current shortcut for that
        in the line edit."""
        # store
        self._isprimary = isprimary
        self._fullname = fullname
        # create intro to show, and store + show it
        tmp = fullname.replace("__", " -> ").replace("_", " ")
        primSec = ["secondary", "primary"][int(isprimary)]
        self._intro = "Set the {} shortcut for:\n{}".format(primSec, tmp)
        self._label.setText(self._intro)
        # set initial value
        if fullname in pyzo.config.shortcuts2:
            current = pyzo.config.shortcuts2[fullname]
            if "," not in current:
                current += ","
            current = current.split(",")
            self._line.setText(current[0] if isprimary else current[1])
    def onClear(self):
        self._line.clear()
        self._line.setFocus()
    def onEdit(self):
        """Test if already in use."""
        # init
        shortcut = self._line.text()
        if not shortcut:
            self._label.setText(self._intro)
            return
        for key in pyzo.config.shortcuts2:
            # get shortcut and test whether it corresponds with what's pressed
            shortcuts = getShortcut(key)
            primSec = ""
            if shortcuts[0].lower() == shortcut.lower():
                primSec = "primary"
            elif shortcuts[1].lower() == shortcut.lower():
                primSec = "secondary"
            # if a correspondence, let the user know
            if primSec and key != self._fullname:
                tmp = "Warning: shortcut already in use for:\n"
                tmp += key.replace("__", " -> ").replace("_", " ")
                self._label.setText(self._intro + "\n\n" + tmp + "\n")
                break
        else:
            self._label.setText(self._intro)
    def onAccept(self):
        shortcut = self._line.text()
        # remove shortcut if present elsewhere
        keys = [key for key in pyzo.config.shortcuts2]  # copy
        for key in keys:
            # get shortcut, test whether it corresponds with what's pressed
            shortcuts = getShortcut(key)
            tmp = list(shortcuts)
            needUpdate = False
            if shortcuts[0].lower() == shortcut.lower():
                tmp[0] = ""
                needUpdate = True
            if shortcuts[1].lower() == shortcut.lower():
                tmp[1] = ""
                needUpdate = True
            if needUpdate:
                tmp = ",".join(tmp)
                tmp = tmp.replace(" ", "")
                if len(tmp) == 1:
                    del pyzo.config.shortcuts2[key]
                else:
                    pyzo.config.shortcuts2[key] = tmp
        # insert shortcut
        if self._fullname:
            # get current and make list of size two
            if self._fullname in pyzo.config.shortcuts2:
                current = list(getShortcut(self._fullname))
            else:
                current = ["", ""]
            # update the list
            current[int(not self._isprimary)] = shortcut
            pyzo.config.shortcuts2[self._fullname] = ",".join(current)
        # close
        self.close()
class KeymappingDialog(QtWidgets.QDialog):
    """The main keymap dialog, it has tabs corresponding with the
    different menus and each tab has a tree representing the structure
    of these menus. The current shortcuts are displayed.
    On double clicking on an item, the shortcut can be edited."""
    def __init__(self, *args):
        QtWidgets.QDialog.__init__(self, *args)
        # set title
        self.setWindowTitle(translate("menu dialog", "Shortcut mappings"))
        # set size
        size = 600, 400
        offset = 0
        size2 = size[0], size[1] + offset
        self.resize(*size2)
        self.setMaximumSize(*size2)
        self.setMinimumSize(*size2)
        self.tab = CompactTabWidget(self, padding=(4, 4, 6, 6))
        self.tab.resize(*size)
        self.tab.move(0, offset)
        self.tab.setMovable(False)
        # fill tab
        self._models = []
        self._trees = []
        for menu in pyzo.main.menuBar()._menus:
            # create treeview and model
            model = KeyMapModel()
            model.setRootMenu(menu)
            tree = QtWidgets.QTreeView(self.tab)
            tree.setModel(model)
            # configure treeview
            tree.clicked.connect(self.onClickSelect)
            tree.doubleClicked.connect(self.onDoubleClick)
            tree.setColumnWidth(0, 150)
            # append to lists
            self._models.append(model)
            self._trees.append(tree)
            self.tab.addTab(tree, menu.title())
        self.tab.currentChanged.connect(self.onTabSelect)
    def closeEvent(self, event):
        # update key setting
        pyzo.keyMapper.keyMappingChanged.emit()
        event.accept()
    def onTabSelect(self):
        pass
    def onClickSelect(self, index):
        # should we show a prompt?
        if index.column():
            self.popupItem(index.internalPointer(), index.column())
    def onDoubleClick(self, index):
        if not index.column():
            self.popupItem(index.internalPointer())
    def popupItem(self, item, shortCutId=1):
        """Popup the dialog to change the shortcut."""
        if isinstance(item, QtWidgets.QAction) and item.text():
            # create prompt dialog
            dlg = KeyMapEditDialog(self)
            dlg.setFullName(item.menuPath, shortCutId == 1)
            # show it
            dlg.exec_()
class AdvancedSettings(QtWidgets.QDialog):
    """Advanced settings
    The Advanced settings dialog contains configuration settings for Pyzo and
plugins.
    Click on an item, to edit settings.
    """
    def __init__(self, *args):
        QtWidgets.QDialog.__init__(self, *args)
        self.conf_file = os.path.join(pyzo.appConfigDir, "config.ssdf")
        self.backup_file = os.path.join(pyzo.appConfigDir, "config.ssdf.bak")
        if not os.path.exists(self.conf_file):
            pyzo.saveConfig()
        if not os.path.exists(self.backup_file):
            self.backupConfig()
        # Set title
        self.setWindowTitle(translate("menu dialog", "Advanced Settings"))
        # Set dialog size
        size = 1000, 600
        offset = 0
        size2 = size[0], size[1] + offset
        self.resize(*size2)
        self.setMinimumSize(*size2)
        # Label
        text = "Before you begin, backup your settings. To modify an existing
setting, click on the value to change it.\nNote that most settings require a
restart for the change to take effect."
        self._label = QtWidgets.QLabel(self)
        self._label.setWordWrap(True)
        self._label.setText(translate("menu dialog", text))
        # Folding buttons
        # fold
        self._btnFold = QtWidgets.QToolButton(self)
        self._btnFold.setIcon(pyzo.icons.text_align_justify)
        self._btnFold.setToolTip("Fold")
        # event
        self._btnFold.clicked.connect(self.btnFoldClicked)
        # unfold
        self._btnUnfold = QtWidgets.QToolButton(self)
        self._btnUnfold.setIcon(pyzo.icons.text_align_right)
        self._btnUnfold.setToolTip("Unfold")
        # event
        self._btnUnfold.clicked.connect(self.btnUnfoldClicked)
        # Search
        self._search = QtWidgets.QLineEdit(self)
        self._search.setPlaceholderText("Search...")
        self._search.setToolTip("Use the search box to find the setting of
interest.")
        # event
        self._search.textChanged.connect(self.searchTextChanged)
        # Tree
        self._tree = QtWidgets.QTreeWidget(self)
        self._tree.setColumnCount(3)
        self._tree.setHeaderLabels(["key", "value", "type"])
        self._tree.setSortingEnabled(True)
        self._tree.sortItems(0, QtCore.Qt.AscendingOrder)
        self._tree.setColumnWidth(0, 300)
        self._tree.setColumnWidth(1, 300)
        # event
        self._tree.itemClicked.connect(self.onClickSelect)
        self._tree.itemChanged.connect(self.currentItemChanged)
        # Backup label
        self._backup_label = QtWidgets.QLabel(self)
        self._backup_label.setText(self.setBackupDate())
        # Buttons
        self._btnFactoryDefault = QtWidgets.QPushButton("Factory Defaults",
self)
        self._btnFactoryDefault.setToolTip("Reset a IDE to its original
settings.")
        self._btnFactoryDefault.clicked.connect(self.btnFactoryDefaultClicked)
        self._btnOverwriteShellSettings = QtWidgets.QPushButton(
            "Overwrite shell settings", self
        )
        self._btnOverwriteShellSettings.setToolTip(
            "Overwrite an existing shell settings with backup shell settings."
        )
self._btnOverwriteShellSettings.clicked.connect(self.replaceShellSettings)
        self._btnBackup = QtWidgets.QPushButton("Backup", self)
        self._btnBackup.setToolTip("Make backup file of the current settings.")
        self._btnBackup.setDefault(True)
        self._btnBackup.clicked.connect(self.btnBackupClicked)
        self._btnRestore = QtWidgets.QPushButton("Restore", self)
        self._btnRestore.setToolTip(
            "Restore settings from last backup file, then restart Pyzo."
        )
        self._btnRestore.clicked.connect(self.btnRestoreClicked)
        # Layouts
        layout_1 = QtWidgets.QHBoxLayout()
        layout_1.addWidget(self._label, 0)
        #
        layout_2 = QtWidgets.QHBoxLayout()
        layout_2.addWidget(self._btnFold, 0)
        layout_2.addWidget(self._btnUnfold, 0)
        layout_2.addWidget(self._search, 0)
        #
        layout_3 = QtWidgets.QVBoxLayout()
        layout_3.addWidget(self._tree, 0)
        layout_3.addWidget(self._backup_label, 0)
        #
        layout_4 = QtWidgets.QHBoxLayout()
        layout_4.addWidget(self._btnFactoryDefault, 0)
        layout_4.addWidget(self._btnOverwriteShellSettings, 0)
        layout_4.addWidget(self._btnBackup, 0)
        layout_4.addWidget(self._btnRestore, 0)
        # Main Layout
        mainLayout = QtWidgets.QVBoxLayout(self)
        mainLayout.addLayout(layout_1, 0)
        mainLayout.addLayout(layout_2, 0)
        mainLayout.addLayout(layout_3, 0)
        mainLayout.addLayout(layout_4, 0)
        mainLayout.setSpacing(2)
        mainLayout.setContentsMargins(4, 4, 4, 4)
        self.setLayout(mainLayout)
        # Fill tree
        self.fillTree()
        self._tree.expandAll()
    def btnFoldClicked(self):
        self._tree.collapseAll()
    def btnUnfoldClicked(self):
        self._tree.expandAll()
    def searchTextChanged(self):
        """As you type hide unmatched settings."""
        # find all matched settings
        find = self._tree.findItems(
            self._search.text(), QtCore.Qt.MatchContains |
QtCore.Qt.MatchRecursive, 0
        )
        items = [i.text(0) for i in find]
        # tree items
        root = self._tree.invisibleRootItem()
        root_count = root.childCount()
        # level 1
        for idx_0 in range(root_count):
            item = root.child(idx_0)
            # level 2
            if item.childCount() > 0:
                for idx_1 in range(item.childCount()):
                    child = item.child(idx_1)
                    hasVisibleChild = False
                    # level 3
                    if child.childCount() > 0:
                        for idx_2 in range(child.childCount()):
                            subchild = child.child(idx_2)
                            # hide options
                            if subchild.text(0) in items:
                                subchild.setHidden(False)
                                hasVisibleChild = True
                            else:
                                subchild.setHidden(True)
                # hide options
                if child.text(0) in items:
                    child.setHidden(False)
                else:
                    if not hasVisibleChild:
                        child.setHidden(True)
    def backupConfig(self):
        """Backup settings."""
        shutil.copyfile(self.conf_file, self.backup_file)
    def setBackupDate(self):
        """Show backup file name and backup date."""
        lastmodified = os.stat(self.backup_file).st_mtime
        datetime.fromtimestamp(lastmodified)
        backup_text = """Backup file: {}, {} """.format(
            self.backup_file, datetime.fromtimestamp(lastmodified)
        )
        return backup_text
    def btnFactoryDefaultClicked(self):
        """Reset a IDE to its original settings."""
        pyzo.saveConfig()
        pyzo.resetConfig()
        pyzo.main.restart()
    def btnBackupClicked(self):
        """Make backup file of the current settings."""
        self.backupConfig()
        self._backup_label.setText(self.setBackupDate())
    def btnRestoreClicked(self):
        """Restore settings from last backup file, then restart Pyzo."""
        pyzo.resetConfig()
        shutil.copyfile(self.backup_file, self.conf_file)
        pyzo.main.restart()
    def currentItemChanged(self, item, column):
        """Save new settings."""
        parent = None
        node = None
        if column == 1:
            # node, parent
            parent = item.parent()
            parent_val = self._tree.indexFromItem(parent, 0).data()
            node = parent.parent()
            node_val = self._tree.indexFromItem(node, 0).data()
            # key, value
            key = self._tree.indexFromItem(item, 0).data()
            value = self._tree.indexFromItem(item, 1).data()
            typ = self._tree.indexFromItem(item, 2).data()
            # convert type
            if typ is not type(value):
                if typ == "<class 'int'>":
                    value = int(value)
                elif typ == "<class 'list'>":
                    value = ast.literal_eval(value)
            # change value
            if node:
                # case: node - parent - key - value
                if node_val in pyzo.config.keys():
                    if (
                        parent_val in pyzo.config[node_val].keys()
                    ):  # bugfix +[node_val]+
                        pyzo.config[node_val][parent_val][key] = value
            else:
                # case: parent - key - value
                if parent:
                    if parent_val in pyzo.config.keys():
                        pyzo.config[parent_val][key] = value
            # bold changed item
            font = item.font(column)
            font.setBold(True)
            item.setFont(column, QtGui.QFont(font))
            # save
            pyzo.saveConfig()
    def fillTree(self):
        """Fill the tree with settings"""
        # fill tree
        for item in pyzo.config.keys():
            root = QtWidgets.QTreeWidgetItem(self._tree, [item])
            node = pyzo.config[item]
            if isinstance(node, pyzo.util.zon.Dict):
                for k, v in node.items():
                    if isinstance(v, pyzo.util.zon.Dict):
                        A = QtWidgets.QTreeWidgetItem(root,
["{}".format(str(k))])
                        for kk, vv in v.items():
                            if isinstance(vv, pyzo.util.zon.Dict):
                                B = QtWidgets.QTreeWidgetItem(A,
["{}".format(str(kk))])
                                for kkk, vvv in vv.items():
                                    QtWidgets.QTreeWidgetItem(
                                        B, [str(kkk), str(vvv), str(type(vvv))]
                                    )
                            else:
                                QtWidgets.QTreeWidgetItem(
                                    A, [str(kk), str(vv), str(type(vv))]
                                )
                    else:
                        QtWidgets.QTreeWidgetItem(root, [str(k), str(v),
str(type(v))])
            elif isinstance(node, list):
                n = 1
                for k in node:
                    if isinstance(k, pyzo.util.zon.Dict):
                        A = QtWidgets.QTreeWidgetItem(root,
["shell_{}".format(str(n))])
                        for kk, vv in k.items():
                            QtWidgets.QTreeWidgetItem(
                                A, [str(kk), str(vv), str(type(vv))]
                            )
                        n += 1
    def onClickSelect(self, item, column):
        """Allow editing only column 1"""
        if column == 1:
            item.setFlags(item.flags() | QtCore.Qt.ItemIsEditable)
            self._tree.editItem(item, column)
    def getBackupShellSettings(self):
        """Extract shell settings from backup file."""
        backup_dict = pyzo.util.zon.load(self.backup_file)
        backup_shell = backup_dict["shellConfigs2"]
        return backup_shell
    def replaceShellSettings(self):
        """Replace current shell setting with backup shell settings."""
        old_shell_settings = self.getBackupShellSettings()
        pyzo.config["shellConfigs2"] = old_shell_settings
        pyzo.saveConfig()
```

```
from pyzo.qt import QtCore, QtGui, QtWidgets
from pyzo import translate
import pyzo
import os
from pyzo.codeeditor import Manager
class PdfExport(QtWidgets.QDialog):
    """
    This class is used to export an editor to a pdf.
    The content of the editor is copied in another editor,
    and then the options chosen are applied by _print()
    """
    def __init__(self):
        super().__init__()
        from pyzo.qt import QtPrintSupport
        self.printer = QtPrintSupport.QPrinter(
            QtPrintSupport.QPrinter.HighResolution,
        )
        # To allow pdf export with color
        self.printer.setColorMode(QtPrintSupport.QPrinter.Color)
        # Default settings
        self.show_line_number = True
        self._enable_syntax_highlighting = True
        # Set title
        self.setWindowTitle(translate("menu dialog", "Pdf Export"))
        # Set dialog size
        size = 1000, 600
        offset = 0
        size2 = size[0], size[1] + offset
        self.resize(*size2)
        # self.setMinimumSize(*size2)
        # Button to export to pdf
        self.validation_button = QtWidgets.QPushButton("Export")
        self.validation_button.clicked.connect(self._export_pdf)
        # Button to update the preview
        self.button_update_preview = QtWidgets.QPushButton("Update preview",
self)
        self.button_update_preview.clicked.connect(self._update_preview)
        # Preview widget
        self.preview = QtPrintSupport.QPrintPreviewWidget(self.printer)
        # Lines numbers option
        self.checkbox_line_number = QtWidgets.QCheckBox(
            "Print line number", self, checked=self.show_line_number
        )
```

```
self.checkbox_line_number.stateChanged.connect(self._get_show_line_number)
        # Make of copy of the editor
        self.current_editor = pyzo.editors.getCurrentEditor()
        self.editor_name = self.current_editor.name
        self.editor_filename = self.current_editor.filename
        self.editor = pyzo.core.editor.PyzoEditor(
            pyzo.editors.getCurrentEditor().toPlainText()
        )
        # Zoom
        # The default zoom is the current zoom used by the editor
        self.original_zoom = pyzo.config.view.zoom
        self.zoom_slider = QtWidgets.QSlider(QtCore.Qt.Horizontal)
        self.zoom_slider.setMinimum(-10)  # Maybe too much ?
        self.zoom_slider.setMaximum(10)
        self.zoom_slider.setTickInterval(1)
        self.zoom_selected = self.original_zoom
        self.zoom_slider.setValue(self.zoom_selected)
        self.zoom_value_label = QtWidgets.QLabel()
        self._zoom_value_changed()
        self.zoom_slider.valueChanged.connect(self._zoom_value_changed)
        # Option for syntax highlighting
        self.checkbox_syntax_highlighting = QtWidgets.QCheckBox(
            "Enable syntax highlighting", self,
checked=self._enable_syntax_highlighting
        )
        self.checkbox_syntax_highlighting.stateChanged.connect(
            self._change_syntax_highlighting_option
        )
        self.combobox_file_name = QtWidgets.QComboBox(self)
        self.combobox_file_name.addItem("Do not print the file name", 0)
        self.combobox_file_name.addItem("Print with file name", 1)
        self.combobox_file_name.addItem("Print with file name and absolute
path", 2)
        self.combobox_file_name.setCurrentIndex(1)
        self.combobox_file_name.setToolTip("The title at the top of the
document")
        # Orientation
        self.combobox_orientation = QtWidgets.QComboBox(self)
        self.combobox_orientation.addItem("Portrait", 0)
        self.combobox_orientation.addItem("Landscape", 1)
        self.combobox_orientation.setToolTip("Orientation of the document")
        # Layout
        self.main_layout = QtWidgets.QHBoxLayout()
        self.setLayout(self.main_layout)
        self.preview.setSizePolicy(
            QtWidgets.QSizePolicy.Expanding, QtWidgets.QSizePolicy.Expanding
        )
        self.right_layout = QtWidgets.QVBoxLayout()
        self.option_layout = QtWidgets.QFormLayout()
        self.main_layout.addWidget(self.preview)
        self.main_layout.addLayout(self.right_layout)
        self.right_layout.addLayout(self.option_layout)
        self.option_layout.addRow(self.combobox_file_name)
        self.option_layout.addRow(self.checkbox_line_number)
        self.option_layout.addRow(self.checkbox_syntax_highlighting)
        self.option_layout.addRow(self.zoom_value_label, self.zoom_slider)
        self.option_layout.addRow(self.combobox_orientation)
        self.bottom_layout = QtWidgets.QHBoxLayout()
        self.right_layout.addLayout(self.bottom_layout)
        self.bottom_layout.addStretch()
        self.bottom_layout.addWidget(self.button_update_preview)
        self.bottom_layout.addWidget(self.validation_button)
        self._update_preview()
    def _print(self):
        """Generate the pdf for preview and export"""
        if self.editor is not None:
            cursor = self.editor.textCursor()
            cursor.movePosition(cursor.Start)
            cursor.movePosition(cursor.End, cursor.KeepAnchor)
            cursor.insertText(pyzo.editors.getCurrentEditor().toPlainText())
            self._set_zoom(self.zoom_selected)
            # Print with line numbers
            lines = self.editor.toPlainText().splitlines()
            nzeros = len(str(len(lines)))
            self._apply_syntax_highlighting()
            starting_line = 0
            self._change_orientation()
            # Print name or filename in the editor
            if self.combobox_file_name.currentIndex():
                starting_line = 1
                if self.combobox_file_name.currentIndex() == 1:
                    lines.insert(0, "# " + self.editor_name + "\n")
                elif self.combobox_file_name.currentIndex() == 2:
                    lines.insert(0, "# " + self.editor_filename + "\n")
            # Print line numbers in the editor
            if self.show_line_number:
                for i in range(starting_line, len(lines)):
                    lines[i] = (
                        str(i + 1 - starting_line).rjust(nzeros, "0") + "| " +
lines[i]
                    )
            cursor = self.editor.textCursor()
            cursor.movePosition(cursor.Start)
            cursor.movePosition(cursor.End, cursor.KeepAnchor)
            cursor.insertText("\n".join(lines))
            # Highlight line numbers
            if self.show_line_number:
                cursor.movePosition(cursor.Start, cursor.MoveAnchor)
                # Move the cursor down 2 lines if a title is printed
                if starting_line != 0:
                    cursor.movePosition(cursor.NextBlock, cursor.MoveAnchor, 2)
                # Apply background for lines numbers
                for i in range(len(lines)):
                    fmt = QtGui.QTextCharFormat()
                    fmt.setBackground(QtGui.QColor(240, 240, 240))
                    cursor.movePosition(cursor.Right, cursor.KeepAnchor, nzeros)
                    cursor.setCharFormat(fmt)
                    cursor.movePosition(cursor.NextBlock, cursor.MoveAnchor)
                    cursor.movePosition(cursor.StartOfBlock, cursor.MoveAnchor)
    def _update_preview(self):
        """Update the widget preview"""
        self._print()
        self.preview.paintRequested.connect(self.editor.print_)
        self.preview.updatePreview()
        self._set_zoom(self.original_zoom)
    def _export_pdf(self):
        """Exports the code as pdf, and opens file manager"""
        if self.editor is not None:
            if True:
                filename = QtWidgets.QFileDialog.getSaveFileName(
                    None, "Export PDF", os.path.expanduser("~"), "*.pdf"
                )
                if isinstance(filename, tuple):  # PySide
                    filename = filename[0]
                if not filename:
                    return
                self.printer.setOutputFileName(filename)
            else:
                d = QtWidgets.QPrintDialog(self.printer)
                d.setWindowTitle("Print code")
                d.setOption(d.PrintSelection,
self.editor.textCursor().hasSelection())
                d.setOption(d.PrintToFile, True)
                ok = d.exec_()
                if ok != d.Accepted:
                    return
        try:
            self._print()
            self.editor.print_(self.printer)
        except Exception as print_error:
            print(print_error)
    def _get_show_line_number(self, state):
        """Change the show_line_number according to the checkbox"""
        if state == QtCore.Qt.Checked:
            self.show_line_number = True
        else:
            self.show_line_number = False
    def _set_zoom(self, value):
        """Apply zoom setting only to the editor used to generate the pdf
        (and the preview)"""
        self.editor.setZoom(pyzo.config.view.zoom + value)
    def _zoom_value_changed(self):
        """Triggered when the zoom slider is changed"""
        self.zoom_selected = self.zoom_slider.value()
        zoom_level = self.zoom_selected - self.zoom_slider.minimum()
        self.zoom_value_label.setText("Zoom level : {}".format(zoom_level))
    def _change_syntax_highlighting_option(self, state):
        """Used for the syntax highlight checkbox when its state change
        to change the option value"""
        if state == QtCore.Qt.Checked:
            self._enable_syntax_highlighting = True
        else:
            self._enable_syntax_highlighting = False
    def _apply_syntax_highlighting(self):
        """Apply the syntax setting when _print() is used"""
        if self._enable_syntax_highlighting:
            text = pyzo.editors.getCurrentEditor().toPlainText()
            ext = os.path.splitext(pyzo.editors.getCurrentEditor()._filename)[1]
            parser = Manager.suggestParser(ext, text)
            self.editor.setParser(parser)
        else:
            self.editor.setParser(None)
    def _change_orientation(self):
        """Set document in portrait or landscape orientation"""
        if hasattr(self.printer, "setOrientation"):  # PySide5, PyQt5
            base = pyzo.qt.QtPrintSupport.QPrinter
            orientation = [base.Portrait, base.Landscape][index]
            self.preview.setOrientation(orientation)
            self.printer.setOrientation(orientation)
        else:  # PySide6, PyQt6
            base = QtGui.QPageLayout
            orientation = [base.Portrait, base.Landscape][index]
            self.preview.setOrientation(orientation)
            layout = QtGui.QPageLayout()
            layout.setOrientation(orientation)
            self.printer.setPageLayout(layout)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module logging
Functionality for logging in pyzo.
"""
import os
import sys, time
import pyzo
# todo: enable logging to a file?
# Define prompts
try:
    sys.ps1
except AttributeError:
    sys.ps1 = ">>> "
try:
    sys.ps2
except AttributeError:
    sys.ps2 = "... "
class DummyStd:
    """For when std is not available."""
    def __init__(self):
        self._closed = False
    def write(self, text):
        pass
    def encoding(self):
        return "utf-8"
    @property
    def closed(self):
        return self._closed
    def close(self):
        self._closed = False
    def flush(self):
        pass
original_print = print
def print(*args, **kwargs):
    # Obtain time string
    t = time.localtime()
    preamble = "{:02g}-{:02g}-{:04g} {:02g}:{:02g}:{:02g}: "
    preamble = preamble.format(
        t.tm_mday, t.tm_mon, t.tm_year, t.tm_hour, t.tm_min, t.tm_sec
    )
    # Prepend to args and print
    args = [preamble] + list(args)
    original_print(*tuple(args), **kwargs)
def splitConsole(stdoutFun=None, stderrFun=None):
    """splitConsole(stdoutFun=None, stderrFun=None)
    Splits the stdout and stderr streams. On each call
    to their write methods, in addition to the original
    write method being called, will call the given
    functions.
    Returns the history of the console (combined stdout
    and stderr).
    Used by the logger shell.
    """
    # Split stdout and stderr
    sys.stdout = OutputStreamSplitter(sys.stdout)
    sys.stderr = OutputStreamSplitter(sys.stderr)
    # Make them share their history
    sys.stderr._history = sys.stdout._history
    # Set defer functions
    if stdoutFun:
        sys.stdout._deferFunction = stdoutFun
    if stderrFun:
        sys.stderr._deferFunction = stderrFun
    # Return history
    return "".join(sys.stdout._history)
class OutputStreamSplitter:
    """This class is used to replace stdout and stderr output
    streams. It defers the stream to the original and to
    a function that can be registered.
    Used by the logger shell.
    """
    def __init__(self, fileObject):
        # Init, copy properties if it was already a splitter
        if isinstance(fileObject, OutputStreamSplitter):
            self._original = fileObject._original
            self._history = fileObject._history
            self._deferFunction = fileObject._deferFunction
        else:
            self._original = fileObject
            self._history = []
            self._deferFunction = self.dummyDeferFunction
        # Replace original with a dummy if None
        if self._original is None:
            self._original = DummyStd()
    def dummyDeferFunction(self, text):
        pass
    def write(self, text):
        """Write method."""
        self._original.write(text)
        self._history.append(text)
        if os.getenv("PYZO_LOG", ""):
            with open(os.getenv("PYZO_LOG"), "at") as f:
                f.write(text)
        try:
            self._deferFunction(text)
        except Exception:
            pass  # self._original.write('error writing to deferred stream')
        # Show in statusbar
        if pyzo.config.view.showStatusbar and len(text) > 1:
            if pyzo.main:
                pyzo.main.statusBar().showMessage(text, 5000)
    def flush(self):
        return self._original.flush()
    @property
    def closed(self):
        return self._original.closed
    def close(self):
        return self._original.close()
    def encoding(self):
        return self._original.encoding()
# Split now, with no defering
splitConsole()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module shell
Defines the shell to be used in pyzo.
This is done in a few inheritance steps:
  - BaseShell inherits BaseTextCtrl and adds the typical shell behaviour.
  - PythonShell makes it specific to Python.
This module also implements ways to communicate with the shell and to run
code in it.
"""
import sys, time
import re
import yoton
import pyzo
from pyzo.util import zon as ssdf  # zon is ssdf-light
from pyzo.qt import QtCore, QtGui, QtWidgets
Qt = QtCore.Qt
from pyzo.codeeditor.highlighter import Highlighter
from pyzo.codeeditor import parsers
from pyzo.core.baseTextCtrl import BaseTextCtrl
from pyzo.core.pyzoLogging import print
from pyzo.core.kernelbroker import KernelInfo, Kernelmanager
from pyzo.core.menu import ShellContextMenu
# Interval for polling messages. Timer for each kernel. I found
# that this one does not affect performance much
POLL_TIMER_INTERVAL = 100  # 100ms 10Hz
# Maximum number of lines in the shell
MAXBLOCKCOUNT = pyzo.config.advanced.shellMaxLines
# todo: we could make command shells to, with autocompletion and coloring...
class YotonEmbedder(QtCore.QObject):
    """Embed the Yoton event loop."""
    def __init__(self):
        QtCore.QObject.__init__(self)
        yoton.app.embed_event_loop(self.postYotonEvent)
    def postYotonEvent(self):
        try:
            QtWidgets.qApp.postEvent(self, QtCore.QEvent(QtCore.QEvent.User))
        except Exception:
            pass  # If pyzo is shutting down, the app may be None
    def customEvent(self, event):
        """This is what gets called by Qt."""
        yoton.process_events(False)
yotonEmbedder = YotonEmbedder()
# Short constants for cursor movement
A_KEEP = QtGui.QTextCursor.KeepAnchor
A_MOVE = QtGui.QTextCursor.MoveAnchor
# Instantiate a local kernel broker upon loading this module
pyzo.localKernelManager = Kernelmanager(public=False)
def finishKernelInfo(info, scriptFile=None):
    """finishKernelInfo(info, scriptFile=None)
    Get a copy of the kernel info struct, with the scriptFile
    and the projectPath set.
    """
    # Make a copy, we do not want to change the original
    info = ssdf.copy(info)
    # Set scriptFile (if '', the kernel will run in interactive mode)
    if scriptFile:
        info.scriptFile = scriptFile
    else:
        info.scriptFile = ""
    # If the file browser is active, and has the check box
    #'add path to Python path' set, set the PROJECTPATH variable
    fileBrowser = pyzo.toolManager.getTool("pyzofilebrowser")
    projectManager = pyzo.toolManager.getTool("pyzoprojectmanager")
    info.projectPath = ""
    if fileBrowser:
        info.projectPath = fileBrowser.getAddToPythonPath()
    if projectManager and not info.projectPath:
        # Only process project manager tool if file browser did not set a path.
        info.projectPath = projectManager.getAddToPythonPath()
    return info
class ShellHighlighter(Highlighter):
    """This highlighter implements highlighting for a shell;
    only the input lines are highlighted with this highlighter.
    """
    def highlightBlock(self, line):
        # Make sure this is a Unicode Python string
        line = str(line)
        # Get previous state
        previousState = self.previousBlockState()
        # Get parser
        parser = None
        if hasattr(self._codeEditor, "parser"):
            parser = self._codeEditor.parser()
        # Get function to get format
        nameToFormat = self._codeEditor.getStyleElementFormat
        # Last line?
        cursor1 = self._codeEditor._cursor1
        cursor2 = self._codeEditor._cursor2
        commandCursor = self._codeEditor._lastCommandCursor
        curBlock = self.currentBlock()
        #
        atLastPrompt, atCurrentPrompt = False, False
        if curBlock.position() == 0:
            pass
        elif curBlock.position() == commandCursor.block().position():
            atLastPrompt = True
        elif curBlock.position() >= cursor1.block().position():
            atCurrentPrompt = True
        if not atLastPrompt and not atCurrentPrompt:
            # Do not highlight anything but current and last prompts
            return
        # Get user data
        bd = self.getCurrentBlockUserData()
        if parser:
            if atCurrentPrompt:
                pos1, pos2 = cursor1.positionInBlock(),
cursor2.positionInBlock()
            else:
                pos1, pos2 = 0, commandCursor.positionInBlock()
            # Check if we should *not* format this line.
            # This is the case for special "executing" text
            # A bit of a hack though ... is there a better way to signal this?
            specialinput = (not atCurrentPrompt) and line[pos2:].startswith(
                "(executing "
            )
            self.setCurrentBlockState(0)
            if specialinput:
                pass  # Let the kernel decide formatting
            else:
                tokens = list(parser.parseLine(line, previousState))
                bd.tokens = tokens
                for token in tokens:
                    # Handle block state
                    if isinstance(token, parsers.BlockState):
                        self.setCurrentBlockState(token.state)
                    else:
                        # Get format
                        try:
                            format = nameToFormat(token.name).textCharFormat
                        except KeyError:
                            # print(repr(nameToFormat(token.name)))
                            continue
                        # Set format
                        # format.setFontWeight(75)
                        if token.start >= pos2:
                            self.setFormat(token.start, token.end - token.start,
format)
            # Set prompt to bold
            if atCurrentPrompt:
                format = QtGui.QTextCharFormat()
                format.setFontWeight(75)
                self.setFormat(pos1, pos2 - pos1, format)
        # Get the indentation setting of the editors
        indentUsingSpaces = self._codeEditor.indentUsingSpaces()
        leadingWhitespace = line[: len(line) - len(line.lstrip())]
        if "\t" in leadingWhitespace and " " in leadingWhitespace:
            # Mixed whitespace
            bd.indentation = 0
            format = QtGui.QTextCharFormat()
            format.setUnderlineStyle(QtGui.QTextCharFormat.SpellCheckUnderline)
            format.setUnderlineColor(QtCore.Qt.red)
            format.setToolTip("Mixed tabs and spaces")
            self.setFormat(0, len(leadingWhitespace), format)
        elif ("\t" in leadingWhitespace and indentUsingSpaces) or (
            " " in leadingWhitespace and not indentUsingSpaces
        ):
            # Whitespace differs from document setting
            bd.indentation = 0
            format = QtGui.QTextCharFormat()
            format.setUnderlineStyle(QtGui.QTextCharFormat.SpellCheckUnderline)
            format.setUnderlineColor(QtCore.Qt.blue)
            format.setToolTip("Whitespace differs from document setting")
            self.setFormat(0, len(leadingWhitespace), format)
        else:
            # Store info for indentation guides
            # amount of tabs or spaces
            bd.indentation = len(leadingWhitespace)
class BaseShell(BaseTextCtrl):
    """The BaseShell implements functionality to make a generic shell."""
    def __init__(self, parent, **kwds):
        super().__init__(
            parent,
            wrap=True,
            showLineNumbers=False,
            showBreakPoints=False,
            highlightCurrentLine=False,
            parser="python",
            **kwds
        )
        # Use a special highlighter that only highlights the input.
        self._setHighlighter(ShellHighlighter)
        # No undo in shells
        self.setUndoRedoEnabled(False)
        # variables we need
        self._more = False
        # We use two cursors to keep track of where the prompt is
        # cursor1 is in front, and cursor2 is at the end of the prompt.
        # They can be in the same position.
        # Further, we store a cursor that selects the last given command,
        # so it can be styled.
        self._cursor1 = self.textCursor()
        self._cursor2 = self.textCursor()
        self._lastCommandCursor = self.textCursor()
        self._lastline_had_cr = False
        self._lastline_had_lf = False
        # When inserting/removing text at the edit line (thus also while typing)
        # keep cursor2 at its place. Only when text is written before
        # the prompt (i.e. in write), this flag is temporarily set to False.
        # Same for cursor1, because sometimes (when there is no prompt) it
        # is at the same position.
        self._cursor1.setKeepPositionOnInsert(True)
        self._cursor2.setKeepPositionOnInsert(True)
        # Similarly, we use the _lastCommandCursor cursor really for pointing.
        self._lastCommandCursor.setKeepPositionOnInsert(True)
        # Variables to keep track of the command history usage
        self._historyNeedle = None  # None means none, "" means look in all
        self._historyStep = 0
        # Set minimum width so 80 lines do fit in smallest font size
        self.setMinimumWidth(200)
        # Hard wrapping. QTextEdit allows hard wrapping at a specific column.
        # Unfortunately, QPlainTextEdit does not.
        self.setWordWrapMode(QtGui.QTextOption.WrapAnywhere)
        # Limit number of lines
        self.setMaximumBlockCount(MAXBLOCKCOUNT)
        # Keep track of position, so we can disable editing if the cursor
        # is before the prompt
        self.cursorPositionChanged.connect(self.onCursorPositionChanged)
        self.setFocusPolicy(Qt.TabFocus)  # See remark at mousePressEvent
    ## Cursor stuff
    def onCursorPositionChanged(self):
        # If the end of the selection (or just the cursor if there is no 
selection)
        # is before the beginning of the line. make the document read-only
        cursor = self.textCursor()
        promptpos = self._cursor2.position()
        if cursor.position() < promptpos or cursor.anchor() < promptpos:
            self.setReadOnly(True)
        else:
            self.setReadOnly(False)
    def ensureCursorAtEditLine(self):
        """
        If the text cursor is before the beginning of the edit line,
        move it to the end of the edit line
        """
        cursor = self.textCursor()
        promptpos = self._cursor2.position()
        if cursor.position() < promptpos or cursor.anchor() < promptpos:
            cursor.movePosition(cursor.End, A_MOVE)  # Move to end of document
            self.setTextCursor(cursor)
            self.onCursorPositionChanged()
    def mousePressEvent(self, event):
        """
        - Disable right MB and middle MB (which pastes by default).
        - Focus policy
            If a user clicks this shell, while it has no focus, we do
            not want the cursor position to change (since generally the
            user clicks the shell to give it the focus). We do this by
            setting the focus-policy to Qt::TabFocus, and we give the
            widget its focus manually from the mousePressedEvent event
            handler
        """
        if not self.hasFocus():
            self.setFocus()
            return
        if event.button() != QtCore.Qt.MidButton:
            super().mousePressEvent(event)
    def contextMenuEvent(self, event):
        """Do not show context menu."""
        pass
    def mouseDoubleClickEvent(self, event):
        BaseTextCtrl.mouseDoubleClickEvent(self, event)
        self._handleClickOnFilename(event.pos())
    def _handleClickOnFilename(self, mousepos):
        """Check whether the text that is clicked is a filename
        and open the file in the editor. If a line number can also be
        detected, open the file at that line number.
        """
        # Get cursor and its current pos
        cursor = self.cursorForPosition(mousepos)
        ocursor = QtGui.QTextCursor(cursor)  # Make a copy to use below
        pos = cursor.positionInBlock()
        # Get line of text for the cursor
        cursor.movePosition(cursor.EndOfBlock, cursor.MoveAnchor)
        cursor.movePosition(cursor.StartOfBlock, cursor.KeepAnchor)
        line = cursor.selectedText()
        if len(line) > 1024:
            return  # safety
        # Get the thing that is clicked, assuming it is delimited with quotes
        line = line.replace("'", '"')
        before = line[:pos].split('"')[-1]
        after = line[pos:].split('"')[0]
        piece = before + after
        if not re.fullmatch(r"<tmp \d+>", piece):
            # Check if it looks like a filename, quit if it does not
            if len(piece) < 4:
                return
            elif not ("/" in piece or "\\" in piece):
                return
            if sys.platform.startswith("win"):
                if piece[1] != ":":
                    return
            else:
                if not piece.startswith("/"):
                    return
        filename = piece
        # Split in parts for getting line number
        line = line[pos + len(after) :]
        line = line.replace(",", " ")
        parts = [p for p in line.split(" ") if p]
        # Iterate over parts
        linenr = None
        for i, part in enumerate(parts):
            if part in ("line", "linenr", "lineno"):
                try:
                    linenr = int(parts[i + 1])
                except IndexError:
                    pass  # no more parts
                except ValueError:
                    pass  # not an integer
                else:
                    break
        # Try again IPython style
        # IPython shows a few lines with the active line indicated by an arrow
        if linenr is None:
            for i in range(4):
                cursor.movePosition(cursor.NextBlock, cursor.MoveAnchor)
                cursor.movePosition(cursor.EndOfBlock, cursor.KeepAnchor)
                line = cursor.selectedText()
                if len(line) > 1024:
                    continue  # safety
                if not line.startswith("-"):
                    continue
                parts = line.split(" ", 2)
                if parts[0] in ("->", "-->", "--->", "---->", "----->"):
                    try:
                        linenr = int(parts[1].strip())
                    except IndexError:
                        pass  # too few parts
                    except ValueError:
                        pass  # not an integer
                    else:
                        break
        # Select word here (in shell)
        cursor = ocursor
        cursor.movePosition(cursor.Left, cursor.MoveAnchor, len(before))
        cursor.movePosition(cursor.Right, cursor.KeepAnchor, len(piece))
        self.setTextCursor(cursor)
        # For syntax errors we have the offset thingy in the file name
        if ".py+" in filename[-9:]:
            filename, _, offset = filename.rpartition("+")
            if linenr is not None:
                try:
                    linenr += int(offset)
                except ValueError:
                    pass
        # Try opening the file (at the line number if we have one)
        result = pyzo.editors.loadFile(filename)
        if result and linenr is not None:
            editor = result._editor
            editor.gotoLine(linenr)
            cursor = editor.textCursor()
            cursor.movePosition(cursor.StartOfBlock)
            cursor.movePosition(cursor.EndOfBlock, cursor.KeepAnchor)
            editor.setTextCursor(cursor)
    ##Indentation: override code editor behaviour
    def indentSelection(self):
        pass
    def dedentSelection(self):
        pass
    ## Key handlers
    def keyPressEvent(self, event):
        if event.key() in [Qt.Key_Return, Qt.Key_Enter]:
            # First check if autocompletion triggered
            if self.potentiallyAutoComplete(event):
                return
            else:
                # Enter: execute line
                # Remove calltip and autocomp if shown
                self.autocompleteCancel()
                self.calltipCancel()
                # reset history needle
                self._historyNeedle = None
                # process
                self.processLine()
                return
        if event.key() == Qt.Key_Escape:
            # Escape clears command
            if not (self.autocompleteActive() or self.calltipActive()):
                self.clearCommand()
        if event.key() == Qt.Key_Home:
            # Home goes to the prompt.
            cursor = self.textCursor()
            if event.modifiers() & Qt.ShiftModifier:
                cursor.setPosition(self._cursor2.position(), A_KEEP)
            else:
                cursor.setPosition(self._cursor2.position(), A_MOVE)
            #
            self.setTextCursor(cursor)
            self.autocompleteCancel()
            return
        if event.key() == Qt.Key_Insert:
            # Don't toggle between insert mode and overwrite mode.
            return True
        # Ensure to not backspace / go left beyond the prompt
        if event.key() in [Qt.Key_Backspace, Qt.Key_Left]:
            self._historyNeedle = None
            if self.textCursor().position() == self._cursor2.position():
                if event.key() == Qt.Key_Backspace:
                    self.textCursor().removeSelectedText()
                return  # Ignore the key, don't go beyond the prompt
        if event.key() in [Qt.Key_Up, Qt.Key_Down] and not
self.autocompleteActive():
            # needle
            if self._historyNeedle is None:
                # get partly-written-command
                #
                # Select text
                cursor = self.textCursor()
                cursor.setPosition(self._cursor2.position(), A_MOVE)
                cursor.movePosition(cursor.End, A_KEEP)
                # Update needle text
                self._historyNeedle = cursor.selectedText()
                self._historyStep = 0
            # Browse through history
            if event.key() == Qt.Key_Up:
                self._historyStep += 1
            else:  # Key_Down
                self._historyStep -= 1
                if self._historyStep < 1:
                    self._historyStep = 1
            # find the command
            c = pyzo.command_history.find_starting_with(
                self._historyNeedle, self._historyStep
            )
            if c is None:
                # found nothing-> reset
                self._historyStep = 0
                c = self._historyNeedle
            # Replace text
            cursor = self.textCursor()
            cursor.setPosition(self._cursor2.position(), A_MOVE)
            cursor.movePosition(cursor.End, A_KEEP)
            cursor.insertText(c)
            self.ensureCursorAtEditLine()
            return
        else:
            # Reset needle
            self._historyNeedle = None
        # if a 'normal' key is pressed, ensure the cursor is at the edit line
        if event.text():
            self.ensureCursorAtEditLine()
        # Default behaviour: BaseTextCtrl
        BaseTextCtrl.keyPressEvent(self, event)
    ## Cut / Copy / Paste / Drag & Drop
    def cut(self):
        """Reimplement cut to only copy if part of the selected text
        is not at the prompt."""
        if self.isReadOnly():
            return self.copy()
        else:
            return BaseTextCtrl.cut(self)
    # def copy(self): # no overload needed
    def paste(self):
        """Reimplement paste to paste at the end of the edit line when
        the position is at the prompt."""
        self.ensureCursorAtEditLine()
        # Paste normally
        return BaseTextCtrl.paste(self)
    def dragEnterEvent(self, event):
        """
        We only support copying of the text
        """
        if event.mimeData().hasText():
            event.setDropAction(QtCore.Qt.CopyAction)
            event.accept()
    def dragMoveEvent(self, event):
        self.dragEnterEvent(event)
    def dropEvent(self, event):
        """
        The shell supports only a single line but the text may contain multiple
        lines. We insert at the editLine only the first non-empty line of the
text
        """
        if event.mimeData().hasText():
            text = event.mimeData().text()
            insertText = ""
            for line in text.splitlines():
                if line.strip():
                    insertText = line
                    break
            # Move the cursor to the position indicated by the drop location,
but
            # ensure it is at the edit line
            self.setTextCursor(self.cursorForPosition(event.pos()))
            self.ensureCursorAtEditLine()
            # Now insert the text
            cursor = self.textCursor()
            cursor.insertText(insertText)
            self.setFocus()
    ## Basic commands to control the shell
    def clearScreen(self):
        """Clear all the previous output from the screen."""
        # Select from beginning of prompt to start of document
        self._cursor1.clearSelection()
        self._cursor1.movePosition(self._cursor1.Start, A_KEEP)  # Keep anchor
        self._cursor1.removeSelectedText()
        # Wrap up
        self.ensureCursorAtEditLine()
        self.ensureCursorVisible()
    def deleteLines(self):
        """Called from the menu option "delete lines", just execute
self.clearCommand()"""
        self.clearCommand()
    def clearCommand(self):
        """Clear the current command, move the cursor right behind
        the prompt, and ensure it's visible.
        """
        # Select from prompt end to length and delete selected text.
        cursor = self.textCursor()
        cursor.setPosition(self._cursor2.position(), A_MOVE)
        cursor.movePosition(cursor.End, A_KEEP)
        cursor.removeSelectedText()
        # Wrap up
        self.ensureCursorAtEditLine()
        self.ensureCursorVisible()
    def _handleBackspaces_split(self, text):
        # while NOT a backspace at first position, or none found
        i = 9999999999999
        while i > 0:
            i = text.rfind("\b", 0, i)
            if i > 0 and text[i - 1] != "\b":
                text = text[0 : i - 1] + text[i + 1 :]
        # Strip the backspaces at the start
        text2 = text.lstrip("\b")
        n = len(text) - len(text2)
        # Done
        return n, text2
    def _handleBackspacesOnList(self, texts):
        """_handleBackspacesOnList(texts)
        Handle backspaces on a list of messages. When printing
        progress, many messages will simply replace each-other, which
        means we can process them much more effectively than when they're
        combined in a list.
        """
        # Init number of backspaces at the start
        N = 0
        for i in range(len(texts)):
            # Remove backspaces in text and how many are left
            n, text = self._handleBackspaces_split(texts[i])
            texts[i] = text
            # Use remaining backspaces to remove backspaces in earlier texts
            while n and i > 0:
                i -= 1
                text = texts[i]
                if len(text) > n:
                    texts[i] = text[:-n]
                    n = 0
                else:
                    texts[i] = ""
                    n -= len(text)
            N += n
        # Insert tabs for start
        if N:
            texts[0] = "\b" * N + texts[0]
        # Return with empy elements popped
        return [t for t in texts if t]
    def _handleBackspaces(self, text):
        """Apply backspaces in the string itself and if there are
        backspaces left at the start of the text, remove the appropriate
        amount of characters from the text.
        Returns the new text.
        """
        # take care of backspaces
        if "\b" in text:
            # Remove backspaces and get how many were at the beginning
            nb, text = self._handleBackspaces_split(text)
            if nb:
                # Select what we remove and delete that
                self._cursor1.clearSelection()
                self._cursor1.movePosition(self._cursor1.Left, A_KEEP, nb)
                self._cursor1.removeSelectedText()
        # Return result
        return text
    def _handleCarriageReturnOnList(self, texts):
        """Discard messages that end with CR and that are not followed
        with LF. Also discard messages followed by a line that starts
        with CR. Assumes that each message is one line.
        """
        if len(texts) < 3:
            # Don't touch texts that might be a single line,
            #  e.g. texts = ['msg', '\r']
            return texts
        for i in range(len(texts) - 1):
            if (
                texts[i].endswith("\r")
                and not texts[i + 1].startswith("\n")
                and not (i > 0 and texts[i - 1].endswith("\n"))
            ) or (
                texts[i + 1].startswith("\r")
                and not texts[i + 1][1:].startswith("\n")
                and not texts[i].endswith("\n")
            ):
                texts[i] = ""
        texts = [t for t in texts if t]
        if len(texts) == 1 and texts[0] == "\r":
            # Never return an isolated carriage return
            texts = []
        return texts
    def _handleCarriageReturn(self, text):
        """Removes the last line if it ended with CR, or if the current new
        message starts with CR.
        Returns the text.
        """
        if "logger" in self.__class__.__name__.lower():
            return text
        # Remove last line if it ended with CR
        cursor = self._cursor1
        if (self._lastline_had_cr and not text.startswith("\n")) or (
            text.startswith("\r")
            and not text[1:].startswith("\n")
            and not self._lastline_had_lf
        ):
            cursor.movePosition(cursor.StartOfLine, cursor.KeepAnchor, 1)
            cursor.removeSelectedText()
        # Is this new line ending in CR?
        self._lastline_had_cr = text.endswith("\r")
        # Is this new line ending in LF?
        self._lastline_had_lf = text.endswith("\n")
        text = text.replace("\r", "")
        return text
    def _splitLinesForPrinting(self, text):
        """Given a text, split the text in lines. Lines that are extremely
        long are split in pieces of 80 characters to increase performance for
        wrapping. This is kind of a failsafe for when the user accidentally
        prints a bitmap or huge list. See issue 98.
        """
        for line in text.splitlines(True):
            if len(line) > 1024:  # about 12 lines of 80 chars
                parts = [line[i : i + 80] for i in range(0, len(line), 80)]
                yield "\n".join(parts)
            else:
                yield line
    def write(self, text, prompt=0, color=None):
        """write(text, prompt=0, color=None)
        Write to the shell. Fauto-ind
        If prompt is 0 (default) the text is printed before the prompt. If
        prompt is 1, the text is printed after the prompt, the new prompt
        becomes null. If prompt is 2, the given text becomes the new prompt.
        The color of the text can also be specified (as a hex-string).
        """
        # From The Qt docs: Note that a cursor always moves when text is
        # inserted before the current position of the cursor, and it always
        # keeps its position when text is inserted after the current position
        # of the cursor.
        # Make sure there's text and make sure its a string
        if not text:
            return
        if isinstance(text, bytes):
            text = text.decode("utf-8")
        # Prepare format
        format = QtGui.QTextCharFormat()
        if color:
            format.setForeground(QtGui.QColor(color))
        # pos1, pos2 = self._cursor1.position(), self._cursor2.position()
        # Just in case, clear any selection of the cursors
        self._cursor1.clearSelection()
        self._cursor2.clearSelection()
        if prompt == 0:
            # Insert text behind prompt (normal streams)
            self._cursor1.setKeepPositionOnInsert(False)
            self._cursor2.setKeepPositionOnInsert(False)
            text = self._handleCarriageReturn(text)
            text = self._handleBackspaces(text)
            self._insertText(self._cursor1, text, format)
        elif prompt == 1:
            # Insert command text after prompt, prompt becomes null (input)
            self._lastCommandCursor.setPosition(self._cursor2.position())
            self._cursor1.setKeepPositionOnInsert(False)
            self._cursor2.setKeepPositionOnInsert(False)
            self._insertText(self._cursor2, text, format)
            self._cursor1.setPosition(self._cursor2.position(), A_MOVE)
        elif prompt == 2 and text == "\b":
            # Remove prompt (used when closing the kernel)
            self._cursor1.setPosition(self._cursor2.position(), A_KEEP)
            self._cursor1.removeSelectedText()
            self._cursor2.setPosition(self._cursor1.position(), A_MOVE)
        elif prompt == 2:
            # Insert text after prompt, inserted text becomes new prompt
            self._cursor1.setPosition(self._cursor2.position(), A_MOVE)
            self._cursor1.setKeepPositionOnInsert(True)
            self._cursor2.setKeepPositionOnInsert(False)
            self._insertText(self._cursor1, text, format)
        # Reset cursor states for the user to type his/her commands
        self._cursor1.setKeepPositionOnInsert(True)
        self._cursor2.setKeepPositionOnInsert(True)
        # Make sure that cursor is visible (only when cursor is at edit line)
        if not self.isReadOnly():
            self.ensureCursorVisible()
        # Scroll along with the text if lines are popped from the top
        elif self.blockCount() == MAXBLOCKCOUNT:
            n = text.count("\n")
            sb = self.verticalScrollBar()
            sb.setValue(sb.value() - n)
    def _insertText(self, cursor, text, format):
        """Insert text at the given cursor, and with the given format.
        This function processes ANSI escape code for formatting and
        colorization: http://en.wikipedia.org/wiki/ANSI_escape_code
        """
        # If necessary, make a new cursor that moves along. We insert
        # the text in pieces, so we need to move along with the text!
        if cursor.keepPositionOnInsert():
            cursor = QtGui.QTextCursor(cursor)
            cursor.setKeepPositionOnInsert(False)
        # Init. We use the solarised color theme
        pattern = r"\x1b\[(.*?)m"
        # CLRS = ['#000', '#F00', '#0F0', '#FF0', '#00F', '#F0F', '#0FF',
'#FFF']
        CLRS = [
            "#657b83",
            "#dc322f",
            "#859900",
            "#b58900",
            "#268bd2",
            "#d33682",
            "#2aa198",
            "#eee8d5",
        ]
        i0 = 0
        for match in re.finditer(pattern, text):
            # Insert pending text with the current format
            # Also update indices
            i1, i2 = match.span()
            cursor.insertText(text[i0:i1], format)
            i0 = i2
            # The formay that we are now going to parse should apply to
            # the text that follow it ...
            # Get parameters
            try:
                params = [int(i) for i in match.group(1).split(";")]
            except ValueError:
                params = []
            if not params:
                params = [0]
            # Process
            for param in params:
                if param == 0:
                    format = QtGui.QTextCharFormat()
                elif param == 1:
                    format.setFontWeight(75)  # Bold
                elif param == 2:
                    format.setFontWeight(25)  # Faint
                elif param == 3:
                    format.setFontItalic(True)  # Italic
                elif param == 4:
                    format.setFontUnderline(True)  # Underline
                #
                elif param == 22:
                    format.setFontWeight(50)  # Not bold or faint
                elif param == 23:
                    format.setFontItalic(False)  # Not italic
                elif param == 24:
                    format.setFontUnderline(False)  # Not underline
                #
                elif 30 <= param <= 37:  # Set foreground color
                    clr = CLRS[param - 30]
                    format.setForeground(QtGui.QColor(clr))
                elif 40 <= param <= 47:
                    pass  # Cannot set background text in QPlainTextEdit
                #
                else:
                    pass  # Not supported
        else:
            # At the end, process the remaining text
            text = text[i0:]
            # Process very long text more efficiently.
            # Insert per line (very long lines are split in smaller ones)
            if len(text) > 1024:
                for line in self._splitLinesForPrinting(text):
                    cursor.insertText(line, format)
            else:
                cursor.insertText(text, format)
    ## Executing stuff
    def processLine(self, line=None, execute=True):
        """processLine(self, line=None, execute=True)
        Process the given line or the current line at the prompt if not given.
        Called when the user presses enter.
        If execute is False will not execute the command. This way
        a message can be written while other ways are used to process
        the command.
        """
        # Can we do this?
        if self.isReadOnly() and not line:
            return
        if line:
            # remove trailing newline(s)
            command = line.rstrip("\n")
        else:
            # Select command
            cursor = self.textCursor()
            cursor.setPosition(self._cursor2.position(), A_MOVE)
            cursor.movePosition(cursor.End, A_KEEP)
            # Sample the text from the prompt and remove it
            command = cursor.selectedText().replace("\u2029", "\n").rstrip("\n")
            cursor.removeSelectedText()
            # Auto-indent. Note: this is rather Python-specific
            command_s = command.lstrip()
            indent = " " * (len(command) - len(command_s))
            if command.strip().endswith(":"):
                indent += "    "
            elif not command_s:
                indent = ""
            if indent:
                cursor.insertText(indent)
            if command:
                # Remember the command in this global history
                pyzo.command_history.append(command)
        if execute:
            command = command.replace("\r\n", "\n")
            self.executeCommand(command + "\n")
    def executeCommand(self, command):
        """Execute the given command.
        Should be overridden.
        """
        # this is a stupid simulation version
        self.write("you executed: " + command + "\n")
        self.write(">>> ", prompt=2)
class PythonShell(BaseShell):
    """The PythonShell class implements the python part of the shell
    by connecting to a remote process that runs a Python interpreter.
    """
    # Emits when the status string has changed or when receiving a new prompt
    stateChanged = QtCore.Signal(BaseShell)
    # Emits when the debug status is changed
    debugStateChanged = QtCore.Signal(BaseShell)
    def __init__(self, parent, info):
        BaseShell.__init__(self, parent)
        # Get standard info if not given.
        if info is None and pyzo.config.shellConfigs2:
            info = pyzo.config.shellConfigs2[0]
        if not info:
            info = KernelInfo(None)
        # Store info so we can reuse it on a restart
        self._info = info
        # For the editor to keep track of attempted imports
        self._importAttempts = []
        # To keep track of the response for introspection
        self._currentCTO = None
        self._currentACO = None
        # Write buffer to store messages in for writing
        self._write_buffer = None
        # Create timer to keep polling any results
        # todo: Maybe use yoton events to process messages as they arrive.
        # I tried this briefly, but it seemd to be less efficient because
        # messages are not so much bach-processed anymore. We should decide
        # on either method.
        self._timer = QtCore.QTimer(self)
        self._timer.setInterval(POLL_TIMER_INTERVAL)  # ms
        self._timer.setSingleShot(False)
        self._timer.timeout.connect(self.poll)
        self._timer.start()
        # Add context menu
        self._menu = ShellContextMenu(shell=self, parent=self)
        self.setContextMenuPolicy(QtCore.Qt.CustomContextMenu)
        self.customContextMenuRequested.connect(
            lambda p: self._menu.popup(self.mapToGlobal(p + QtCore.QPoint(0,
3)))
```

```
        )
        # Keep track of breakpoints
        pyzo.editors.breakPointsChanged.connect(self.sendBreakPoints)
        # Start!
        self.resetVariables()
        self.connectToKernel(info)
    def resetVariables(self):
        """Resets some variables."""
        # Reset read state
        self.setReadOnly(False)
        # Variables to store state, python version, builtins and keywords
        self._state = ""
        self._debugState = {}
        self._version = ""
        self._builtins = []
        self._keywords = []
        self._startup_info = {}
        self._start_time = 0
        # (re)set import attempts
        self._importAttempts[:] = []
        # Update
        self.stateChanged.emit(self)
    def connectToKernel(self, info):
        """connectToKernel()
        Create kernel and connect to it.
        """
        # Create yoton context
        self._context = ct = yoton.Context()
        # Create stream channels
        self._strm_out = yoton.SubChannel(ct, "strm-out")
        self._strm_err = yoton.SubChannel(ct, "strm-err")
        self._strm_raw = yoton.SubChannel(ct, "strm-raw")
        self._strm_echo = yoton.SubChannel(ct, "strm-echo")
        self._strm_prompt = yoton.SubChannel(ct, "strm-prompt")
        self._strm_broker = yoton.SubChannel(ct, "strm-broker")
        self._strm_action = yoton.SubChannel(ct, "strm-action", yoton.OBJECT)
        # Set channels to sync mode. This means that if the pyzo cannot process
        # the messages fast enough, the sending side is blocked for a short
        # while. We don't want our users to miss any messages.
        for c in [self._strm_out, self._strm_err]:
            c.set_sync_mode(True)
        # Create control channels
        self._ctrl_command = yoton.PubChannel(ct, "ctrl-command")
        self._ctrl_code = yoton.PubChannel(ct, "ctrl-code", yoton.OBJECT)
        self._ctrl_broker = yoton.PubChannel(ct, "ctrl-broker")
        # Create status channels
        self._stat_interpreter = yoton.StateChannel(ct, "stat-interpreter")
        self._stat_cd = yoton.StateChannel(ct, "stat-cd")
        self._stat_debug = yoton.StateChannel(ct, "stat-debug", yoton.OBJECT)
        self._stat_startup = yoton.StateChannel(ct, "stat-startup",
yoton.OBJECT)
        self._stat_breakpoints = yoton.StateChannel(
            ct, "stat-breakpoints", yoton.OBJECT
        )
        self._stat_startup.received.bind(self._onReceivedStartupInfo)
        # Create introspection request channel
        self._request = yoton.ReqChannel(ct, "reqp-introspect")
        # Connect! The broker will only start the kernel AFTER
        # we connect, so we do not miss out on anything.
        slot = pyzo.localKernelManager.createKernel(finishKernelInfo(info))
        self._brokerConnection = ct.connect("localhost:%i" % slot)
        self._brokerConnection.closed.bind(self._onConnectionClose)
        # Force updating of breakpoints
        pyzo.editors.updateBreakPoints()
        # todo: see polling vs events
    #         # Detect incoming messages
    #         for c in [self._strm_out, self._strm_err, self._strm_raw,
    #                 self._strm_echo, self._strm_prompt, self._strm_broker,
    #                 self._strm_action,
    #                 self._stat_interpreter, self._stat_debug]:
    #             c.received.bind(self.poll)
    def get_kernel_cd(self):
        """Get current working dir of kernel."""
        return self._stat_cd.recv()
    def _onReceivedStartupInfo(self, channel):
        startup_info = channel.recv()
        # Store the whole dict
        self._startup_info = startup_info
        # Store when we received this
        self._start_time = time.time()
        # Set version
        version = startup_info.get("version", None)
        if isinstance(version, tuple):
            if version < (3,):
                self.setParser("python2")
            else:
                self.setParser("python3")
            version = [str(v) for v in version]
            self._version = ".".join(version[:2])
        # Set keywords
        L = startup_info.get("keywords", None)
        if isinstance(L, list):
            self._keywords = L
        # Set builtins
        L = startup_info.get("builtins", None)
        if isinstance(L, list):
            self._builtins = L
        # Notify
        self.stateChanged.emit(self)
    ## Introspection processing methods
    def processCallTip(self, cto):
        """Processes a calltip request using a CallTipObject instance."""
        # Try using buffer first (not if we're not the requester)
        if self is cto.textCtrl:
            if cto.tryUsingBuffer():
                return
        # Clear buffer to prevent doing a second request
        # and store cto to see whether the response is still wanted.
        cto.setBuffer("")
        self._currentCTO = cto
        # Post request
        future = self._request.signature(cto.name)
        future.add_done_callback(self._processCallTip_response)
        future.cto = cto
    def _processCallTip_response(self, future):
        """Process response of shell to show signature."""
        # Process future
        if future.cancelled():
            # print('Introspect cancelled')  # No kernel
            return
        elif future.exception():
            print("Introspect-exception: ", future.exception())
            return
        else:
            response = future.result()
            cto = future.cto
        # First see if this is still the right editor (can also be a shell)
        editor1 = pyzo.editors.getCurrentEditor()
        editor2 = pyzo.shells.getCurrentShell()
        if cto.textCtrl not in [editor1, editor2]:
            # The editor or shell starting the autocomp is no longer active
            cto.textCtrl.autocompleteCancel()
            return
        # Invalid response
        if response is None:
            cto.textCtrl.autocompleteCancel()
            return
        # If still required, show tip, otherwise only store result
        if cto is self._currentCTO:
            cto.finish(response)
        else:
            cto.setBuffer(response)
    def processAutoComp(self, aco):
        """Processes an autocomp request using an AutoCompObject instance."""
        # Try using buffer first (not if we're not the requester)
        if self is aco.textCtrl:
            if aco.tryUsingBuffer():
                return
        # Include builtins and keywords?
        if not aco.name:
            aco.addNames(self._builtins)
            if pyzo.config.settings.autoComplete_keywords:
                aco.addNames(self._keywords)
        # Set buffer to prevent doing a second request
        # and store aco to see whether the response is still wanted.
        aco.setBuffer()
        self._currentACO = aco
        # Post request
        future = self._request.dir(aco.name)
        future.add_done_callback(self._processAutoComp_response)
        future.aco = aco
    def _processAutoComp_response(self, future):
        """Process the response of the shell for the auto completion."""
        # Process future
        if future.cancelled():
            # print('Introspect cancelled') # No living kernel
            return
        elif future.exception():
            print("Introspect-exception: ", future.exception())
            return
        else:
            response = future.result()
            aco = future.aco
        # First see if this is still the right editor (can also be a shell)
        editor1 = pyzo.editors.getCurrentEditor()
        editor2 = pyzo.shells.getCurrentShell()
        if aco.textCtrl not in [editor1, editor2]:
            # The editor or shell starting the autocomp is no longer active
            aco.textCtrl.autocompleteCancel()
            return
        # Add result to the list
        foundNames = []
        if response is not None:
            foundNames = response
        aco.addNames(foundNames)
        # Process list
        if aco.name and not foundNames:
            # No names found for the requested name. This means
            # it does not exist, let's try to import it
            importNames, importLines = pyzo.parser.getFictiveImports(editor1)
            baseName = aco.nameInImportNames(importNames)
            if baseName:
                line = importLines[baseName].strip()
                if line not in self._importAttempts:
                    # Do import
                    self.processLine(line + " # auto-import")
                    self._importAttempts.append(line)
                    # Wait a barely noticable time to increase the chances
                    # That the import is complete when we repost the request.
                    time.sleep(0.2)
                    # To be sure, decrease the experiration date on the buffer
                    aco.setBuffer(timeout=1)
                    # Repost request
                    future = self._request.signature(aco.name)
                    future.add_done_callback(self._processAutoComp_response)
                    future.aco = aco
        else:
            # If still required, show list, otherwise only store result
            if self._currentACO is aco:
                aco.finish()
            else:
                aco.setBuffer()
    ## Methods for executing code
    def executeCommand(self, text):
        """executeCommand(text)
        Execute one-line command in the remote Python session.
        """
        # Ensure edit line is selected (to reset scrolling to end)
        self.ensureCursorAtEditLine()
        self._ctrl_command.send(text)
    def executeCode(self, text, fname, lineno=None, cellName=None, 
changeDir=False):
        """executeCode(text, fname, lineno, cellName=None)
        Execute (run) a large piece of code in the remote shell.
        text: the source code to execute
        filename: the file from which the source comes
        lineno: the first lineno of the text in the file, where 0 would be
        the first line of the file...
        The text (source code) is first pre-processed:
        - convert all line-endings to \n
        - remove all empty lines at the end
        - remove commented lines at the end
        - convert tabs to spaces
        - dedent so minimal indentation is zero
        """
        # Convert tabs to spaces
        text = text.replace("\t", " " * 4)
        # Make sure there is always *some* text
        if not text:
            text = " "
        if lineno is None:
            lineno = 0
            cellName = fname  # run all
        # Examine the text line by line...
        # - check for empty/commented lined at the end
        # - calculate minimal indentation
        lines = text.splitlines()
        lastLineOfCode = 0
        minIndent = 99
        for linenr in range(len(lines)):
            # Get line
            line = lines[linenr]
            # Check if empty (can be commented, but nothing more)
            tmp = line.split("#", 1)[0]  # get part before first #
            if tmp.count(" ") == len(tmp):
                continue  # empty line, proceed
            else:
                lastLineOfCode = linenr
            # Calculate indentation
            tmp = line.lstrip(" ")
            indent = len(line) - len(tmp)
            if indent < minIndent:
                minIndent = indent
        # Copy all proper lines to a new list,
        # remove minimal indentation, but only if we then would only remove
        # spaces (in the case of commented lines)
        lines2 = []
        for linenr in range(lastLineOfCode + 1):
            line = lines[linenr]
            # Remove indentation,
            if line[:minIndent].count(" ") == minIndent:
                line = line[minIndent:]
            else:
                line = line.lstrip(" ")
            lines2.append(line)
        # Ensure edit line is selected (to reset scrolling to end)
        self.ensureCursorAtEditLine()
        # Send message
        text = "\n".join(lines2)
        msg = {
            "source": text,
            "fname": fname,
            "lineno": lineno,
            "cellName": cellName,
            "changeDir": int(changeDir),
        }
        self._ctrl_code.send(msg)
    def sendBreakPoints(self, breaks):
        """Send all breakpoints."""
        # breaks is a dict of filenames to integers
        self._stat_breakpoints.send(breaks)
    ## The polling methods and terminating methods
    def poll(self, channel=None):
        """poll()
        To keep the shell up-to-date.
        Call this periodically.
        """
        if self._write_buffer:
            # There is still data in the buffer
            sub, M = self._write_buffer
        else:
            # Check what subchannel has the latest message pending
            sub = yoton.select_sub_channel(
                self._strm_out,
                self._strm_err,
                self._strm_echo,
                self._strm_raw,
                self._strm_broker,
                self._strm_prompt,
            )
            # Read messages from it
            if sub:
                M = sub.recv_selected()
                # M = [sub.recv()] # Slow version (for testing)
                # Optimization: handle backspaces on stack of messages
                if sub is self._strm_out:
                    M = self._handleCarriageReturnOnList(M)
                    M = self._handleBackspacesOnList(M)
            # New prompt?
            if sub is self._strm_prompt:
                self.stateChanged.emit(self)
        # Write all pending messages that are later than any other message
        if sub:
            # Select messages to process
            N = 256
            M, buffer = M[:N], M[N:]
            # Buffer the rest
            if buffer:
                self._write_buffer = sub, buffer
            else:
                self._write_buffer = None
            # Get how to deal with prompt
            prompt = 0
            if sub is self._strm_echo:
                prompt = 1
            elif sub is self._strm_prompt:
                prompt = 2
            # Get color
            color = None
            if sub is self._strm_broker:
                color = "#000"
            elif sub is self._strm_raw:
                color = "#888888"  # Halfway
            elif sub is self._strm_err:
                color = "#F00"
            # Write
            self.write("".join(M), prompt, color)
        # Do any actions?
        action = self._strm_action.recv(False)
        if action:
            if action == "cls":
                self.clearScreen()
            elif action.startswith("open "):
                parts = action.split(" ")
                parts.pop(0)
                try:
                    linenr = int(parts[0])
                    parts.pop(0)
                except ValueError:
                    linenr = None
                fname = " ".join(parts)
                editor = pyzo.editors.loadFile(fname)
                if editor and linenr:
                    editor._editor.gotoLine(linenr)
            else:
                print("Unkown action: %s" % action)
        # ----- status
        # Do not update status when the kernel is not really up and running
        # self._version is set when the startup info is received
        if not self._version:
            return
        # Update status
        state = self._stat_interpreter.recv()
        if state != self._state:
            self._state = state
            self.stateChanged.emit(self)
        # Update debug status
        state = self._stat_debug.recv()
        if state != self._debugState:
            self._debugState = state
            self.debugStateChanged.emit(self)
    def interrupt(self):
        """interrupt()
        Send a Keyboard interrupt signal to the main thread of the
        remote process.
        """
        # Ensure edit line is selected (to reset scrolling to end)
        self.ensureCursorAtEditLine()
        self._ctrl_broker.send("INT")
    def restart(self, scriptFile=None):
        """restart(scriptFile=None)
        Terminate the shell, after which it is restarted.
        Args can be a filename, to execute as a script as soon as the
        shell is back up.
        """
        # Ensure edit line is selected (to reset scrolling to end)
        self.ensureCursorAtEditLine()
        # Get info
        info = finishKernelInfo(self._info, scriptFile)
        # Create message and send
        msg = "RESTART\n" + ssdf.saves(info)
        self._ctrl_broker.send(msg)
        # Reset
        self.resetVariables()
    def terminate(self):
        """terminate()
        Terminates the python process. It will first try gently, but
        if that does not work, the process shall be killed.
        To be notified of the termination, connect to the "terminated"
        signal of the shell.
        """
        # Ensure edit line is selected (to reset scrolling to end)
        self.ensureCursorAtEditLine()
        self._ctrl_broker.send("TERM")
    def closeShell(self):  # do not call it close(); that is a reserved method.
        """closeShell()
        Very simple. This closes the shell. If possible, we will first
        tell the broker to terminate the kernel.
        The broker will be cleaned up if there are no clients connected
        and if there is no active kernel. In a multi-user environment,
        we should thus be able to close the shell without killing the
        kernel. But in a closed 1-to-1 environment we really want to
        prevent loose brokers and kernels dangling around.
        In both cases however, it is the responsibility of the broker to
        terminate the kernel, and the shell will simply assume that this
        will work :)
        """
        # If we can, try to tell the broker to terminate the kernel
        if self._context and self._context.connection_count:
            self.terminate()
            self._context.flush()  # Important, make sure the message is send!
            self._context.close()
        # Adios
        pyzo.shells.removeShell(self)
    def _onConnectionClose(self, c, why):
        """To be called after disconnecting.
        In general, the broker will not close the connection, so it can
        be considered an error-state if this function is called.
        """
        # Stop context
        if self._context:
            self._context.close()
        # New (empty prompt)
        self._cursor1.movePosition(self._cursor1.End, A_MOVE)
        self._cursor2.movePosition(self._cursor2.End, A_MOVE)
        self.write("\n\n")
        self.write("Lost connection with broker:\n")
        self.write(why)
        self.write("\n\n")
        # Set style to indicate dead-ness
        self.setReadOnly(True)
        # Goto end such that the closing message is visible
        cursor = self.textCursor()
        cursor.movePosition(cursor.End, A_MOVE)
        self.setTextCursor(cursor)
        self.ensureCursorVisible()
if __name__ == "__main__":
    b = BaseShell(None)
    b.show()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module shellInfoDialog
Implements shell configuration dialog.
"""
import os, sys
from pyzo.qt import QtCore, QtGui, QtWidgets  # noqa
import pyzo
from pyzo.core.pyzoLogging import print
from pyzo.core.kernelbroker import KernelInfo
from pyzo import translate
## Implement widgets that have a common interface
class ShellInfoLineEdit(QtWidgets.QLineEdit):
    def setTheText(self, value):
        self.setText(value)
    def getTheText(self):
        return self.text()
class ShellInfo_name(ShellInfoLineEdit):
    def __init__(self, *args, **kwargs):
        ShellInfoLineEdit.__init__(self, *args, **kwargs)
        self.editingFinished.connect(self.onValueChanged)
        t = translate("shell", "name ::: The name of this configuration.")
        self.setPlaceholderText(t.tt)
    def setTheText(self, value):
        ShellInfoLineEdit.setTheText(self, value)
        self.onValueChanged()
    def onValueChanged(self):
        self.parent().parent().parent().setTabTitle(self.getTheText())
class ShellInfo_exe(QtWidgets.QComboBox):
    def __init__(self, *args):
        QtWidgets.QComboBox.__init__(self, *args)
    def _interpreterName(self, p):
        if p.is_conda:
            return "%s  [v%s, conda]" % (p.path, p.version)
        else:
            return "%s  [v%s]" % (p.path, p.version)
    def setTheText(self, value):
        # Init
        self.clear()
        self.setEditable(True)
        self.setInsertPolicy(self.InsertAtTop)
        # Get known interpreters from shellDialog (which are sorted by version)
        shellDialog = self
        while not isinstance(shellDialog, ShellInfoDialog):
            shellDialog = shellDialog.parent()
        interpreters = shellDialog.interpreters
        exes = [p.path for p in interpreters]
        # Hande current value
        if value in exes:
            value = self._interpreterName(interpreters[exes.index(value)])
        else:
            self.addItem(value)
        # Add all found interpreters
        for p in interpreters:
            self.addItem(self._interpreterName(p))
        # Set current text
        self.setEditText(value)
    def getTheText(self):
        # return self.currentText().split('(')[0].rstrip()
        value = self.currentText()
        if value.endswith("]") and "[" in value:
            value = value.rsplit("[", 1)[0]
        return value.strip()
class ShellInfo_ipython(QtWidgets.QCheckBox):
    def __init__(self, parent):
        QtWidgets.QCheckBox.__init__(self, parent)
        t = translate("shell", "ipython ::: Use IPython shell if available.")
        self.setText(t.tt)
        self.setChecked(False)
    def setTheText(self, value):
        if value.lower() in [
            "",
            "no",
            "false",
        ]:  # Also for empty string; default is False
            self.setChecked(False)
        else:
            self.setChecked(True)
    def getTheText(self):
        if self.isChecked():
            return "yes"
        else:
            return "no"
```

```
class ShellInfo_gui(QtWidgets.QComboBox):
    # For (backward) compatibility
    COMPAT = {"QT4": "PYQT4"}
    # GUI names
    GUIS = [
        ("None", "no GUI support"),
        ("Auto", "Use what is available (recommended)"),
        ("Asyncio", "Python's builtin event loop"),
        ("PySide6", "LGPL licensed wrapper to Qt6"),
        ("PySide2", "LGPL licensed wrapper to Qt5"),
        ("PySide", "LGPL licensed wrapper to Qt4"),
        ("PyQt6", "GPL/commercial licensed wrapper to Qt6"),
        ("PyQt5", "GPL/commercial licensed wrapper to Qt5"),
        ("PyQt4", "GPL/commercial licensed wrapper to Qt4"),
        ("Tornado", "Tornado asynchronous networking library"),
        ("Tk", "Tk widget toolkit"),
        ("WX", "wxPython"),
        ("FLTK", "The fast light toolkit"),
        ("GTK", "GIMP Toolkit"),
    ]
    # GUI descriptions
    def setTheText(self, value):
        # Process value
        value = value.upper()
        value = self.COMPAT.get(value, value)
        # Set options
        ii = 0
        self.clear()
        for i in range(len(self.GUIS)):
            gui, des = self.GUIS[i]
            if value == gui.upper():
                ii = i
            self.addItem("%s  -  %s" % (gui, des))
        # Set current text
        self.setCurrentIndex(ii)
    def getTheText(self):
        text = self.currentText().lower()
        return text.partition("-")[0].strip()
class ShellinfoWithSystemDefault(QtWidgets.QVBoxLayout):
    DISABLE_SYSTEM_DEFAULT = sys.platform == "darwin"
    SYSTEM_VALUE = ""
    def __init__(self, parent, widget):
        # Do not pass parent, because is a sublayout
        QtWidgets.QVBoxLayout.__init__(self)
        # Layout
        self.setSpacing(1)
        self.addWidget(widget)
        # Create checkbox widget
        if not self.DISABLE_SYSTEM_DEFAULT:
            t = translate("shell", "Use system default")
            self._check = QtWidgets.QCheckBox(t, parent)
            self._check.stateChanged.connect(self.onCheckChanged)
            self.addWidget(self._check)
        # The actual value of this shell config attribute
        self._value = ""
        # A buffered version, so that clicking the text box does not
        # remove the value at once
        self._bufferedValue = ""
    def onEditChanged(self):
        if self.DISABLE_SYSTEM_DEFAULT or not self._check.isChecked():
            self._value = self.getWidgetText()
    def onCheckChanged(self, state):
        if state:
            self._bufferedValue = self._value
            self.setTheText(self.SYSTEM_VALUE)
        else:
            self.setTheText(self._bufferedValue)
    def setTheText(self, value):
        if self.DISABLE_SYSTEM_DEFAULT:
            # Just set the value
            self._edit.setReadOnly(False)
            self.setWidgetText(value)
        elif value != self.SYSTEM_VALUE:
            # Value given, enable edit
            self._check.setChecked(False)
            self._edit.setReadOnly(False)
            # Set the text
            self.setWidgetText(value)
        else:
            # Use system default, disable edit widget
            self._check.setChecked(True)
            self._edit.setReadOnly(True)
            # Set text using system environment
            self.setWidgetText(None)
        # Store value
        self._value = value
    def getTheText(self):
        return self._value
class ShellInfo_pythonPath(ShellinfoWithSystemDefault):
    SYSTEM_VALUE = "$PYTHONPATH"
    def __init__(self, parent):
        # Create sub-widget
        self._edit = QtWidgets.QTextEdit(parent)
        self._edit.zoomOut(1)
        self._edit.setMaximumHeight(80)
        self._edit.setMinimumWidth(200)
        self._edit.textChanged.connect(self.onEditChanged)
        # Instantiate
        ShellinfoWithSystemDefault.__init__(self, parent, self._edit)
    def getWidgetText(self):
        return self._edit.toPlainText()
    def setWidgetText(self, value=None):
        if value is None:
            pp = os.environ.get("PYTHONPATH", "")
            pp = pp.replace(os.pathsep, "\n").strip()
            value = "$PYTHONPATH:\n%s\n" % pp
        self._edit.setText(value)
# class ShellInfo_startupScript(ShellinfoWithSystemDefault):
#
#     SYSTEM_VALUE = '$PYTHONSTARTUP'
#
#     def __init__(self, parent):
#
#         # Create sub-widget
#         self._edit = QtWidgets.QLineEdit(parent)
#         self._edit.textEdited.connect(self.onEditChanged)
#
#         # Instantiate
#         ShellinfoWithSystemDefault.__init__(self, parent, self._edit)
#
#
#     def getWidgetText(self):
#         return self._edit.text()
#
#
#     def setWidgetText(self, value=None):
#         if value is None:
#             pp = os.environ.get('PYTHONSTARTUP','').strip()
#             if pp:
```

```
#                 value = '$PYTHONSTARTUP: "%s"' % pp
#             else:
#                 value = '$PYTHONSTARTUP: None'
#
#         self._edit.setText(value)
class ShellInfo_startupScript(QtWidgets.QVBoxLayout):
    DISABLE_SYSTEM_DEFAULT = sys.platform == "darwin"
    SYSTEM_VALUE = "$PYTHONSTARTUP"
    RUN_AFTER_GUI_TEXT = "# AFTER_GUI - code below runs after integrating the
GUI\n"
    def __init__(self, parent):
        # Do not pass parent, because is a sublayout
        QtWidgets.QVBoxLayout.__init__(self)
        # Create sub-widget
        self._edit1 = QtWidgets.QLineEdit(parent)
        self._edit1.textEdited.connect(self.onEditChanged)
        if sys.platform.startswith("win"):
            self._edit1.setPlaceholderText("C:\\path\\to\\script.py")
        else:
            self._edit1.setPlaceholderText("/path/to/script.py")
        #
        self._edit2 = QtWidgets.QTextEdit(parent)
        self._edit2.zoomOut(1)
        self._edit2.setMaximumHeight(80)
        self._edit2.setMinimumWidth(200)
        self._edit2.textChanged.connect(self.onEditChanged)
        # Layout
        self.setSpacing(1)
        self.addWidget(self._edit1)
        self.addWidget(self._edit2)
        # Create radio widget for system default
        t = translate("shell", "Use system default")
        self._radio_system = QtWidgets.QRadioButton(t, parent)
        self._radio_system.toggled.connect(self.onCheckChanged)
        self.addWidget(self._radio_system)
        if self.DISABLE_SYSTEM_DEFAULT:
            self._radio_system.hide()
        # Create radio widget for file
        t = translate("shell", "File to run at startup")
        self._radio_file = QtWidgets.QRadioButton(t, parent)
        self._radio_file.toggled.connect(self.onCheckChanged)
        self.addWidget(self._radio_file)
        # Create radio widget for code
        t = translate("shell", "Code to run at startup")
        self._radio_code = QtWidgets.QRadioButton(t, parent)
        self._radio_code.toggled.connect(self.onCheckChanged)
        self.addWidget(self._radio_code)
        # The actual value of this shell config attribute
        self._value = ""
        # A buffered version, so that clicking the text box does not
        # remove the value at once
        self._valueFile = ""
        self._valueCode = "\n"
    def onEditChanged(self):
        if self._radio_file.isChecked():
            self._value = self._valueFile = self._edit1.text().strip()
        elif self._radio_code.isChecked():
            # ensure newline!
            self._value = self._valueCode = self._edit2.toPlainText().strip() +
"\n"
    def onCheckChanged(self, state):
        if self._radio_system.isChecked():
            self.setWidgetText(self.SYSTEM_VALUE)
        elif self._radio_file.isChecked():
            self.setWidgetText(self._valueFile)
        elif self._radio_code.isChecked():
            self.setWidgetText(self._valueCode)
    def setTheText(self, value):
        self.setWidgetText(value, True)
        self._value = value
    def setWidgetText(self, value, init=False):
        self._value = value
        if value == self.SYSTEM_VALUE and not self.DISABLE_SYSTEM_DEFAULT:
            # System default
            if init:
                self._radio_system.setChecked(True)
            pp = os.environ.get("PYTHONSTARTUP", "").strip()
            if pp:
                value = '$PYTHONSTARTUP: "%s"' % pp
            else:
                value = "$PYTHONSTARTUP: None"
            #
            self._edit1.setReadOnly(True)
            self._edit1.show()
            self._edit2.hide()
            self._edit1.setText(value)
        elif "\n" not in value:
            # File
            if init:
                self._radio_file.setChecked(True)
            self._edit1.setReadOnly(False)
            self._edit1.show()
            self._edit2.hide()
            self._edit1.setText(value)
        else:
            # Code
            if init:
                self._radio_code.setChecked(True)
            self._edit1.hide()
            self._edit2.show()
            if not value.strip():
                value = self.RUN_AFTER_GUI_TEXT
            self._edit2.setText(value)
    def getTheText(self):
        return self._value
class ShellInfo_startDir(ShellInfoLineEdit):
    def __init__(self, parent):
        ShellInfoLineEdit.__init__(self, parent)
        if sys.platform.startswith("win"):
            self.setPlaceholderText("C:\\path\\to\\your\\python\\modules")
        else:
            self.setPlaceholderText("/path/to/your/python/modules")
class ShellInfo_argv(ShellInfoLineEdit):
    def __init__(self, parent):
        ShellInfoLineEdit.__init__(self, parent)
        self.setPlaceholderText('arg1 arg2 "arg with spaces"')
class ShellInfo_environ(QtWidgets.QTextEdit):
    EXAMPLE = "PYZO_PROCESS_EVENTS_WHILE_DEBUGGING=1\nEXAMPLE_VAR1=value1"
    def __init__(self, parent):
        QtWidgets.QTextEdit.__init__(self, parent)
        self.zoomOut(1)
        self.setPlaceholderText(self.EXAMPLE)
    def _cleanText(self, txt):
        return "\n".join([line.strip() for line in txt.splitlines()])
    def setTheText(self, value):
        value = self._cleanText(value)
        self.setText(value)
    def getTheText(self):
        value = self.toPlainText()
        value = self._cleanText(value)
        return value
## The dialog class and container with tabs
class ShellInfoTab(QtWidgets.QScrollArea):
    INFO_KEYS = [
        translate("shell", "name ::: The name of this configuration."),
        translate("shell", "exe ::: The Python executable."),
        translate("shell", "ipython ::: Use IPython shell if available."),
        translate(
            "shell",
            "gui ::: The GUI toolkit to integrate (for interactive plotting,
etc.).",
        ),
        translate(
            "shell",
            "pythonPath ::: A list of directories to search for modules and
packages. Write each path on a new line, or separate with the default seperator
for this OS.",
        ),  # noqa
        translate(
            "shell",
            "startupScript ::: The script to run at startup (not in script
mode).",
        ),
        translate("shell", "startDir ::: The start directory (not in script
mode)."),
        translate("shell", "argv ::: The command line arguments (sys.argv)."),
        translate("shell", "environ ::: Extra environment variables
(os.environ)."),
    ]
    def __init__(self, parent):
        QtWidgets.QScrollArea.__init__(self, parent)
        # Init the scroll area
        self.setHorizontalScrollBarPolicy(QtCore.Qt.ScrollBarAlwaysOff)
        self.setVerticalScrollBarPolicy(QtCore.Qt.ScrollBarAsNeeded)
        self.setWidgetResizable(True)
        self.setFrameShape(QtWidgets.QFrame.NoFrame)
        # Create widget and a layout
        self._content = QtWidgets.QWidget(parent)
        self._formLayout = QtWidgets.QFormLayout(self._content)
        # Collect classes of widgets to instantiate
        classes = []
        for t in self.INFO_KEYS:
            className = "ShellInfo_" + t.key
            cls = globals()[className]
            classes.append((t, cls))
        # Instantiate all classes
        self._shellInfoWidgets = {}
        for t, cls in classes:
            # Instantiate and store
            instance = cls(self._content)
            self._shellInfoWidgets[t.key] = instance
            # Create label
            label = QtWidgets.QLabel(t, self._content)
            label.setToolTip(t.tt)
            # Add to layout
            self._formLayout.addRow(label, instance)
        # Add delete button
        t = translate("shell", "Delete ::: Delete this shell configuration")
        label = QtWidgets.QLabel("", self._content)
        instance = QtWidgets.QPushButton(pyzo.icons.cancel, t, self._content)
        instance.setToolTip(t.tt)
        instance.setAutoDefault(False)
        instance.clicked.connect(self.parent().parent().onTabClose)
        deleteLayout = QtWidgets.QHBoxLayout()
        deleteLayout.addWidget(instance, 0)
        deleteLayout.addStretch(1)
        # Add to layout
        self._formLayout.addRow(label, deleteLayout)
        # Apply layout
        self._formLayout.setSpacing(15)
        self._content.setLayout(self._formLayout)
        self.setWidget(self._content)
    def setTabTitle(self, name):
        tabWidget = self.parent().parent()
        tabWidget.setTabText(tabWidget.indexOf(self), name)
    def setInfo(self, info=None):
        """Set the shell info struct, and use it to update the widgets.
        Not via init, because this function also sets the tab name.
        """
        # If info not given, use default as specified by the KernelInfo struct
        if info is None:
            info = KernelInfo()
            # Name
            n = self.parent().parent().count()
            if n > 1:
                info.name = "Shell config %i" % n
        # Store info
        self._info = info
        # Set widget values according to info
        try:
            for key in info:
                widget = self._shellInfoWidgets.get(key, None)
                if widget is not None:
                    widget.setTheText(info[key])
        except Exception as why:
            print("Error setting info in shell config:", why)
            print(info)
    def getInfo(self):
        info = self._info
        # Set struct values according to widgets
        try:
            for key, widget in self._shellInfoWidgets.items():
                info[key] = widget.getTheText()
        except Exception as why:
            print("Error getting info in shell config:", why)
            print(info)
        # Return the original (but modified) ssdf Dict object
        return info
class ShellInfoDialog(QtWidgets.QDialog):
    """Dialog to edit the shell configurations."""
    def __init__(self, *args):
        QtWidgets.QDialog.__init__(self, *args)
        self.setModal(True)
        # Set title
        self.setWindowTitle(pyzo.translate("shell", "Shell configurations"))
        # Create tab widget
        self._tabs = QtWidgets.QTabWidget(self)
        # self._tabs = CompactTabWidget(self, padding=(4,4,5,5))
        # self._tabs.setDocumentMode(False)
        self._tabs.setMovable(True)
        # Get known interpreters (sorted them by version)
        # Do this here so we only need to do it once ...
        from pyzo.util.interpreters import get_interpreters
        self.interpreters = list(reversed(get_interpreters("2.4")))
        # Introduce an entry if there's none
        if not pyzo.config.shellConfigs2:
            w = ShellInfoTab(self._tabs)
            self._tabs.addTab(w, "---")
            w.setInfo()
        # Fill tabs
        for item in pyzo.config.shellConfigs2:
            w = ShellInfoTab(self._tabs)
            self._tabs.addTab(w, "---")
            w.setInfo(item)
        # Enable making new tabs and closing tabs
        self._add = QtWidgets.QToolButton(self)
        self._tabs.setCornerWidget(self._add)
        self._add.clicked.connect(self.onAdd)
        self._add.setToolButtonStyle(QtCore.Qt.ToolButtonTextBesideIcon)
        self._add.setIcon(pyzo.icons.add)
        self._add.setText(translate("shell", "Add config"))
        #
        # self._tabs.setTabsClosable(True)
        self._tabs.tabCloseRequested.connect(self.onTabClose)
        # Create buttons
        cancelBut = QtWidgets.QPushButton("Cancel", self)
        okBut = QtWidgets.QPushButton("Done", self)
        cancelBut.clicked.connect(self.close)
        okBut.clicked.connect(self.applyAndClose)
        # Layout for buttons
        buttonLayout = QtWidgets.QHBoxLayout()
        buttonLayout.addStretch(1)
        buttonLayout.addWidget(cancelBut)
        buttonLayout.addSpacing(10)
        buttonLayout.addWidget(okBut)
        # Layout the widgets
        mainLayout = QtWidgets.QVBoxLayout(self)
        mainLayout.addSpacing(8)
        mainLayout.addWidget(self._tabs, 0)
        mainLayout.addLayout(buttonLayout, 0)
        self.setLayout(mainLayout)
        # Prevent resizing
        self.show()
        self.setMinimumSize(500, 400)
        self.resize(640, 500)
        # self.setMaximumHeight(500)
    def onAdd(self):
        # Create widget and add to tabs
        w = ShellInfoTab(self._tabs)
        self._tabs.addTab(w, "---")
        w.setInfo()
        # Select
        self._tabs.setCurrentWidget(w)
        w.setFocus()
    def onTabClose(self):
        index = self._tabs.currentIndex()
        self._tabs.removeTab(index)
    def applyAndClose(self, event=None):
        self.apply()
        self.close()
    def apply(self):
        """Apply changes for all tabs."""
        # Clear
        pyzo.config.shellConfigs2 = []
        # Set new versions. Note that although we recreate the list,
        # the list is filled with the orignal structs, so having a
        # reference to such a struct (as the shell has) will enable
        # you to keep track of any made changes.
        for i in range(self._tabs.count()):
            w = self._tabs.widget(i)
            pyzo.config.shellConfigs2.append(w.getInfo())
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module shellStack
Implements the stack of shells. Also implements the nifty debug button
and a dialog to edit the shell configurations.
"""
import time
import webbrowser
from pyzo.qt import QtCore, QtGui, QtWidgets  # noqa
import pyzo
from pyzo import translate
from pyzo.core.shell import PythonShell
from pyzo.core.pyzoLogging import print  # noqa
from pyzo.core.menu import ShellTabContextMenu, ShellButtonMenu
from pyzo.core.icons import ShellIconMaker
def shellTitle(shell, moreinfo=False):
    """Given a shell instance, build the text title to represent it."""
    # Get name
    nameText = shell._info.name
    # Build version text
    if shell._version:
        versionText = "v{}".format(shell._version)
    else:
        versionText = "v?"
    # Build gui text
    guiText = shell._startup_info.get("gui")
    guiText = guiText or ""
    if guiText.lower() in ["none", ""]:
        guiText = "without gui"
    else:
        guiText = "with " + guiText + " gui"
    # Build state text
    stateText = shell._state or ""
    # Build text for elapsed time
    elapsed = time.time() - shell._start_time
    hh = elapsed // 3600
    mm = (elapsed - hh * 3600) // 60
    ss = elapsed - hh * 3600 - mm * 60
    runtimeText = "runtime: %i:%02i:%02i" % (hh, mm, ss)
    # Build text
    if not moreinfo:
        text = nameText
    else:
        text = "'%s' (%s %s) - %s, %s" % (
            nameText,
            versionText,
            guiText,
            stateText,
            runtimeText,
        )
    # Done
    return text
class ShellStackWidget(QtWidgets.QWidget):
    """The shell stack widget provides a stack of shells.
    It wrapps a QStackedWidget that contains the shell objects. This
    stack is used as a reference to synchronize the shell selection with.
    We keep track of what is the current selected shell and apply updates
    if necessary. Therefore, changing the current shell in the stack
    should be enough to invoke a full update.
    """
    # When the current shell changes.
    currentShellChanged = QtCore.Signal()
    # When the current shells state (or debug state) changes,
    # or when a new prompt is received.
    # Also fired when the current shell changes.
    currentShellStateChanged = QtCore.Signal()
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        # create toolbar
        self._toolbar = QtWidgets.QToolBar(self)
        self._toolbar.setMaximumHeight(26)
        self._toolbar.setIconSize(QtCore.QSize(16, 16))
        # create stack
        self._stack = QtWidgets.QStackedWidget(self)
        # Populate toolbar
        self._shellButton = ShellControl(self._toolbar, self._stack)
        self._debugmode = 0
        self._dbs = DebugStack(self._toolbar)
        #
        self._toolbar.addWidget(self._shellButton)
        self._toolbar.addSeparator()
        # self._toolbar.addWidget(self._dbc) -> delayed, see addContextMenu()
        self._interpreterhelp = InterpreterHelper(self)
        # widget layout
        layout = QtWidgets.QVBoxLayout()
        layout.setSpacing(0)
        # set margins
        margin = pyzo.config.view.widgetMargin
        layout.setContentsMargins(margin, margin, margin, margin)
        layout.addWidget(self._toolbar)
        layout.addWidget(self._stack, 0)
        layout.addWidget(self._interpreterhelp, 0)
        self.setLayout(layout)
        # make callbacks
        self._stack.currentChanged.connect(self.onCurrentChanged)
        self.showInterpreterHelper()
    def __iter__(self):
        i = 0
        while i < self._stack.count():
            w = self._stack.widget(i)
            i += 1
            yield w
    def showInterpreterHelper(self, show=True):
        self._interpreterhelp.setVisible(show)
        self._toolbar.setVisible(not show)
        self._stack.setVisible(not show)
        if show:
            self._interpreterhelp.detect()
    def addShell(self, shellInfo=None):
        """addShell()
        Add a shell to the widget."""
        # Create shell and add to stack
        shell = PythonShell(self, shellInfo)
        self._stack.addWidget(shell)
        # Bind to signals
        shell.stateChanged.connect(self.onShellStateChange)
        shell.debugStateChanged.connect(self.onShellDebugStateChange)
        # Select it and focus on it (invokes onCurrentChanged)
        self._stack.setCurrentWidget(shell)
        shell.setFocus()
        return shell
    def removeShell(self, shell):
        """removeShell()
        Remove an existing shell from the widget
        """
        self._stack.removeWidget(shell)
    def onCurrentChanged(self, index):
        """When another shell is selected, update some things."""
        # Get current
        shell = self.getCurrentShell()
        # Call functions
        self.onShellStateChange(shell)
        self.onShellDebugStateChange(shell)
        # Emit Signal
        self.currentShellChanged.emit()
    def onShellStateChange(self, shell):
        """Called when the shell state changes, and is called
        by onCurrentChanged. Sets the mainwindow's icon if busy.
        """
        # Keep shell button and its menu up-to-date
        self._shellButton.updateShellMenu(shell)
        if shell is self.getCurrentShell():  # can be None
            # Update application icon
            if shell and shell._state in ["Busy"]:
                pyzo.main.setWindowIcon(pyzo.iconRunning)
            else:
                pyzo.main.setWindowIcon(pyzo.icon)
            # Send signal
            self.currentShellStateChanged.emit()
    def onShellDebugStateChange(self, shell):
        """Called when the shell debug state changes, and is called
        by onCurrentChanged. Sets the debug button.
        """
        if shell is self.getCurrentShell():
            # Update debug info
            if shell and shell._debugState:
                info = shell._debugState
                self._debugmode = info["debugmode"]
                for action in self._debugActions:
                    action.setEnabled(self._debugmode == 2)
                self._debugActions[-1].setEnabled(self._debugmode > 0)  # Stop
                self._dbs.setTrace(shell._debugState)
            else:
                for action in self._debugActions:
                    action.setEnabled(False)
                self._debugmode = 0
                self._dbs.setTrace(None)
            # Send signal
            self.currentShellStateChanged.emit()
    def getCurrentShell(self):
        """getCurrentShell()
        Get the currently active shell.
        """
        w = None
        if self._stack.count():
            w = self._stack.currentWidget()
        if not w:
            return None
        else:
            return w
    def getShells(self):
        """Get all shell in stack as list"""
        shells = []
        for i in range(self._stack.count()):
            shell = self.getShellAt(i)
            if shell is not None:
                shells.append(shell)
        return shells
    def getShellAt(self, i):
        return
        """ Get shell at current tab index """
        return self._stack.widget(i)
    def addContextMenu(self):
        # A bit awkward... but the ShellMenu needs the ShellStack, so it
        # can only be initialized *after* the shellstack is created ...
        # Give shell tool button a menu
        self._shellButton.setMenu(ShellButtonMenu(self, "Shell button menu"))
        self._shellButton.menu().aboutToShow.connect(
            self._shellButton._elapsedTimesTimer.start
        )
        # Also give it a context menu
        self._shellButton.setContextMenuPolicy(QtCore.Qt.CustomContextMenu)
self._shellButton.customContextMenuRequested.connect(self.contextMenuTriggered)
        # Add actions
        for action in pyzo.main.menuBar()._menumap["shell"]._shellActions:
            action = self._toolbar.addAction(action)
        self._toolbar.addSeparator()
        # Add debug actions
        self._debugActions = []
        for action in pyzo.main.menuBar()._menumap["shell"]._shellDebugActions:
            self._debugActions.append(action)
            action = self._toolbar.addAction(action)
        # Delayed-add debug control buttons
        self._toolbar.addWidget(self._dbs)
    def contextMenuTriggered(self, p):
        """Called when context menu is clicked"""
        # Get index of shell belonging to the tab
        shell = self.getCurrentShell()
        if shell:
            p =
self._shellButton.mapToGlobal(self._shellButton.rect().bottomLeft())
            ShellTabContextMenu(shell=shell, parent=self).popup(p)
    def onShellAction(self, action):
        shell = self.getCurrentShell()
        if shell:
            getattr(shell, action)()
class ShellControl(QtWidgets.QToolButton):
    """A button that can be used to select a shell and start a new shell."""
    def __init__(self, parent, shellStack):
        QtWidgets.QToolButton.__init__(self, parent)
        # Store reference of shell stack
        self._shellStack = shellStack
        # Keep reference of actions corresponding to shells
        self._shellActions = []
        # Set text and tooltip
        self.setText("Warming up ...")
        self.setToolTip(translate("shells", "Click to select shell."))
        self.setToolButtonStyle(QtCore.Qt.ToolButtonTextBesideIcon)
        self.setPopupMode(self.InstantPopup)
        # Set icon
        self._iconMaker = ShellIconMaker(self)
        self._iconMaker.updateIcon("busy")  # Busy initializing
        # Create timer
        self._elapsedTimesTimer = QtCore.QTimer(self)
        self._elapsedTimesTimer.setInterval(1000)
        self._elapsedTimesTimer.setSingleShot(False)
        self._elapsedTimesTimer.timeout.connect(self.onElapsedTimesTimer)
    def updateShellMenu(self, shellToUpdate=None):
        """Update the shell menu. Ensure that there is a menu item
        for each shell. If shellToUpdate is given, updates the corresponding
        menu item.
        """
        menu = self.menu()
        # Get shells now active
        currentShell = self._shellStack.currentWidget()
        shells = [self._shellStack.widget(i) for i in
range(self._shellStack.count())]
        # Synchronize actions. Remove invalid actions
        for action in self._shellActions:
            # Check match with shells
            if action._shell in shells:
                shells.remove(action._shell)
            else:
                menu.removeAction(action)
            # Update checked state
            if action._shell is currentShell and currentShell:
                action.setChecked(True)
            else:
                action.setChecked(False)
            # Update text if necessary
            if action._shell is shellToUpdate:
                action.setText(shellTitle(shellToUpdate, True))
        # Any items left in shells need a menu item
        # Dont give them an icon, or the icon is used as checkbox thingy
        for shell in shells:
            text = shellTitle(shell)
            action = menu.addItem(text, None, self._shellStack.setCurrentWidget,
shell)
            action._shell = shell
            action.setCheckable(True)
            self._shellActions.append(action)
        # Is the shell being updated the current?
        if currentShell is shellToUpdate and currentShell is not None:
            self._iconMaker.updateIcon(currentShell._state)
            self.setText(shellTitle(currentShell))
        elif currentShell is None:
            self._iconMaker.updateIcon("")
            self.setText("No shell selected")
    def onElapsedTimesTimer(self):
        # Automatically turn timer off is menu is hidden
        if not self.menu().isVisible():
            self._elapsedTimesTimer.stop()
            return
        # Update text for each shell action
        for action in self._shellActions:
            action.setText(shellTitle(action._shell, True))
# todo: remove this?
# class DebugControl(QtWidgets.QToolButton):
```

```
#     """ A button to control debugging.
#     """
#
#     def __init__(self, parent):
#         QtWidgets.QToolButton.__init__(self, parent)
#
#         # Flag
#         self._debugmode = False
#
#         # Set text
#         self.setText(translate('debug', 'Debug'))
#         self.setIcon(pyzo.icons.bug)
#         self.setToolButtonStyle(QtCore.Qt.ToolButtonTextBesideIcon)
#         #self.setPopupMode(self.InstantPopup)
#
#         # Bind to triggers
#         self.triggered.connect(self.onTriggered)
#         self.pressed.connect(self.onPressed)
#         self.buildMenu()
#
#
#     def buildMenu(self):
#
#         # Count breakpoints
#         bpcount = 0
#         for e in pyzo.editors:
#             bpcount += len(e.breakPoints())
#
#         # Prepare a text
#         clearallbps = translate('debug', 'Clear all {} breakpoints')
#         clearallbps = clearallbps.format(bpcount)
#
#         # Set menu
#         menu = QtWidgets.QMenu(self)
#         self.setMenu(menu)
#
#         for cmd, enabled, icon, text in [
#                 ('CLEAR', self._debugmode==0, pyzo.icons.bug_delete,
clearallbps),
#                 ('PM', self._debugmode==0, pyzo.icons.bug_error,
#                     translate('debug', 'Postmortem: debug from last
traceback')),
#                 ('STOP', self._debugmode>0, pyzo.icons.debug_quit,
#                     translate('debug', 'Stop debugging')),
# #                 ('NEXT', self._debugmode==2, pyzo.icons.debug_next,
# #                     translate('debug', 'Next: proceed until next line')),
# #                 ('STEP', self._debugmode==2, pyzo.icons.debug_step,
# #                     translate('debug', 'Step: proceed one step')),
# #                 ('RETURN', self._debugmode==2, pyzo.icons.debug_return,
# #                     translate('debug', 'Return: proceed until returns')),
# #                 ('CONTINUE', self._debugmode==2, pyzo.icons.debug_continue,
# #                     translate('debug', 'Continue: proceed to next 
breakpoint')),
#                 ]:
#             if cmd is None:
#                 menu.addSeparator()
#             else:
#                 if icon is not None:
#                     a = menu.addAction(icon, text)
#                 else:
#                     a = menu.addAction(text)
#                 if hasattr(text, 'tt'):
#                     a.setToolTip(text.tt)
#                 a.cmd = cmd
#                 a.setEnabled(enabled)
#
#
#     def onPressed(self, show=True):
#         self.buildMenu()
#         self.showMenu()
#
#
#     def onTriggered(self, action):
#         if action.cmd == 'PM':
#             # Initiate postmortem debugging
#             shell = pyzo.shells.getCurrentShell()
#             if shell:
#                 shell.executeCommand('DB START\n')
#
#         elif action.cmd == 'CLEAR':
#             # Clear all breakpoints
#             for e in pyzo.editors:
#                 e.clearBreakPoints()
#
#         else:
#             command = action.cmd.upper()
#             shell = pyzo.shells.getCurrentShell()
#             if shell:
#                 shell.executeCommand('DB %s\n' % command)
#
#
#     def setTrace(self, info):
#         """ Determine whether we are in debug mode.
#         """
#         if info is None:
#             self._debugmode = 0
#         else:
#             self._debugmode = info['debugmode']
class DebugStack(QtWidgets.QToolButton):
    """A button that shows the stack trace."""
    def __init__(self, parent):
        QtWidgets.QToolButton.__init__(self, parent)
        # Set text and tooltip
        self._baseText = translate("debug", "Stack")
        self.setText("%s:" % self._baseText)
        self.setIcon(pyzo.icons.text_align_justify)
        self.setToolButtonStyle(QtCore.Qt.ToolButtonTextBesideIcon)
        self.setPopupMode(self.InstantPopup)
        # Bind to triggers
        self.triggered.connect(self.onTriggered)
    def onTriggered(self, action):
        # Get shell
        shell = pyzo.shells.getCurrentShell()
        if not shell:
            return
        # Change stack index
        if not action._isCurrent:
            shell.executeCommand("DB FRAME {}\n".format(action._index))
        # Open file and select line
        if True:
            line = action.text().split(": ", 1)[1]
            self.debugFocus(line)
    def setTrace(self, info):
        """Set the stack trace. This method is called from
        the shell that receives the trace via its status channel
        directly from the interpreter.
        If trace is None, removes the trace
        """
        # Get info
        if info:
            index, frames, debugmode = info["index"], info["frames"],
info["debugmode"]
        else:
            index, frames = -1, []
        if (not frames) or (debugmode == 0):
            # Remove trace
            self.setMenu(None)
            self.setText("")  # (self._baseText)
            self.setEnabled(False)
            pyzo.editors.setDebugLineIndicators(None)
        else:
            # Get the current frame
            theAction = None
            # Create menu and add __main__
            menu = QtWidgets.QMenu(self)
            self.setMenu(menu)
            # Fill trace
            for i in range(len(frames)):
                thisIndex = i + 1
                # Set text for action
                text = '{}: File "{}", line {}, in {}'
                text = text.format(thisIndex, *frames[i])
                action = menu.addAction(text)
                action._index = thisIndex
                action._isCurrent = False
                if thisIndex == index:
                    action._isCurrent = True
                    theAction = action
                    self.debugFocus(text.split(": ", 1)[1])  # Load editor
            # Get debug indicators
            debugIndicators = []
            for i in range(len(frames)):
                thisIndex = i + 1
                filename, linenr, func = frames[i]
                debugIndicators.append((filename, linenr))
                if thisIndex == index:
                    break
            # Set debug indicators
            pyzo.editors.setDebugLineIndicators(*debugIndicators)
            # Highlight current item and set the button text
            if theAction:
                menu.setDefaultAction(theAction)
                # self.setText(theAction.text().ljust(20))
                i = theAction._index
                text = "{} ({}/{}):  ".format(self._baseText, i, len(frames))
                self.setText(text)
            self.setEnabled(True)
    def debugFocus(self, lineFromDebugState):
        """debugFocus(lineFromDebugState)
        Open the file and show the linenr of the given lineFromDebugState.
        """
        # Get filenr and item
        try:
            tmp = lineFromDebugState.split(", in ")[0].split(", line ")
            filename = tmp[0][len("File ") :].strip('"')
            linenr = int(tmp[1].strip())
        except Exception:
            return "Could not focus!"
        # Cannot open <console>
        if filename == "<console>":
            return "Stack frame is <console>."
        elif filename.startswith("<ipython-input-"):
            return "Stack frame is IPython input."
        # Go there!
        result = pyzo.editors.loadFile(filename)
        if not result:
            return "Could not open file where the error occured."
        else:
            editor = result._editor
            # Goto line and select it
            editor.gotoLine(linenr)
            cursor = editor.textCursor()
            cursor.movePosition(cursor.StartOfBlock)
            cursor.movePosition(cursor.EndOfBlock, cursor.KeepAnchor)
            editor.setTextCursor(cursor)
class InterpreterHelper(QtWidgets.QWidget):
    """This sits in place of a shell to help the user download miniconda."""
    def __init__(self, parent):
        super().__init__(parent)
        self._label = QtWidgets.QLabel("hello world")
        self._label.setTextFormat(QtCore.Qt.RichText)
        self._label.setWordWrap(True)
        # self._label.setOpenExternalLinks(True)
        self._label.linkActivated.connect(self.handle_link)
        font = self._label.font()
        font.setPointSize(font.pointSize() + 2)
        self._label.setFont(font)
        layout = QtWidgets.QVBoxLayout()
        self.setLayout(layout)
        layout.addWidget(self._label, 1)
    def refresh(self):
        self._label.setText("Detecting interpreters ...")
        QtWidgets.qApp.sendPostedEvents()
        QtWidgets.qApp.processEvents()
        self.detect()
    def detect(self):
        python_link = '<a href="https://www.python.org/">Python</a>'
        conda_link = '<a href="https://miniconda.pyzo.org">Miniconda</a>'
        self._the_exe = None
        configs = pyzo.config.shellConfigs2
        # Hide now?
        if configs and configs[0].exe:
            self._label.setText("Happy coding!")
            QtCore.QTimer.singleShot(1200, self.hide_this)
            return
        # Try to find an interpreter
        from pyzo.util.interpreters import get_interpreters
        interpreters = list(reversed(get_interpreters("2.4")))
        conda_interpreters = [i for i in interpreters if i.is_conda]
        conda_interpreters.sort(key=lambda x: len(x.path.replace("pyzo", "pyzo"
* 10)))
        # Always sleep for a bit, so show that we've refreshed
        time.sleep(0.05)
        if conda_interpreters and conda_interpreters[0].version > "3":
            self._the_exe = conda_interpreters[0].path
            text = """Pyzo detected a conda environment in:
                      <br />%s<br /><br />
                      You can <a
href='usefound'>use&nbsp;this&nbsp;environment</a>
                      (recommended), or manually specify an interpreter
                      by setting the exe in the <a
href='config'>shell&nbsp;config</a>.
                      <br /><br />Click one of the links above, or <a
href='refresh'>refresh</a>.
                   """ % (
                self._the_exe,
            )
        elif interpreters and interpreters[0].version > "3":
            self._the_exe = interpreters[0].path
            text = """Pyzo detected a Python interpreter in:
                      <br />%s<br /><br />
                      You can <a
href='usefound'>use&nbsp;this&nbsp;environment</a>
                      (recommended), or manually specify an interpreter
                      by setting the exe in the <a
href='config'>shell&nbsp;config</a>.
                      <br /><br />Click one of the links above, or <a
href='refresh'>refresh</a>.
                   """ % (
                self._the_exe,
            )
        elif interpreters:
            text = """Pyzo detected a Python interpreter,
                      but it is Python 2. We strongly recommend using Python 3
instead.
                      <br /><br />
                      If you installed %s or %s in a non-default location,
                      or if you want to manually specify an interpreter,
                      set the exe in the <a href='config'>shell&nbsp;config</a>.
                      <br /><br />Click one of the links above, or <a
href='refresh'>refresh</a>.
                   """ % (
                python_link,
                conda_link,
            )
        else:
            text = """Pyzo did not detect any Python interpreters.
                      We recomment installing %s or %s
                      (and click <a href='refresh'>refresh</a> when done).
                      <br /><br />
                      If you installed Python or Miniconda in a non-default
location,
                      or if you want to manually specify the interpreter,
                      set the exe in the <a href='config'>shell&nbsp;config</a>.
                   """ % (
                python_link,
                conda_link,
            )
        link_style = "font-weight: bold; color:#369; text-decoration:underline;"
        self._label.setText(text.replace("<a ", '<a style="%s" ' % link_style))
    def handle_link(self, url):
        if url == "refresh":
            self.refresh()
        elif url == "config":
            self.editShellConfig()
        elif url == "usefound":
            self.useFound()
        elif url.startswith(("http://", "https://")):
            webbrowser.open(url)
        else:
            raise ValueError("Unknown link in conda helper: %s" % url)
    def editShellConfig(self):
        from pyzo.core.shellInfoDialog import ShellInfoDialog
        d = ShellInfoDialog()
        d.exec_()
        self.refresh()
        self.restart_shell()
    def useFound(self):
        # Set newfound interpreter
        if self._the_exe:
            configs = pyzo.config.shellConfigs2
            if not configs:
                from pyzo.core.kernelbroker import KernelInfo
                pyzo.config.shellConfigs2.append(KernelInfo())
            configs[0].exe = self._the_exe
            self.restart_shell()
        self.refresh()
    def hide_this(self):
        shells = self.parent()
        shells.showInterpreterHelper(False)
    def restart_shell(self):
        shells = self.parent()
        shell = shells.getCurrentShell()
        if shell is not None:
            shell.closeShell()
        shells.addShell(pyzo.config.shellConfigs2[0])
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module splash
Defines splash window shown during startup.
"""
import os
import pyzo
from pyzo.qt import QtCore, QtGui, QtWidgets  # noqa
from pyzo import translate
STYLESHEET = """
QWidget {
    background-color: #268bd2;
}
QFrame {
    background-image: url("%s");
    background-repeat: no-repeat;
    background-position: center;
}
QLabel {
    color: #222;
    background: #46abf2;
    border-radius:20px;
}
"""
splash_text = """
<p>{text_title}</p>
<p>{text_version} {version}</p>
<p>{text_os} <a href='http://pyzo.org/'>http://pyzo.org</a></p>
"""
class LogoWidget(QtWidgets.QFrame):
    def __init__(self, parent):
        QtWidgets.QFrame.__init__(self, parent)
        self.setMinimumSize(256, 256)
        self.setMaximumSize(256, 256)
class LabelWidget(QtWidgets.QWidget):
    def __init__(self, parent, distro=None):
        QtWidgets.QWidget.__init__(self, parent)
        self.setMinimumSize(360, 256)  # Ensure title fits nicely
        # Create label widget and costumize
        self._label = QtWidgets.QLabel(self)
        self._label.setTextFormat(QtCore.Qt.RichText)
        self._label.setOpenExternalLinks(True)
        self._label.setWordWrap(True)
        self._label.setMargin(20)
        # Set font size (absolute value)
        font = self._label.font()
        font.setPointSize(11)  # (font.pointSize()+1)
        self._label.setFont(font)
        # Build
        text_title = translate(
            "splash", "This is <b>Pyzo</b><br />the Python IDE for scientific
computing"
        )
        text_version = translate("splash", "Version")
        text_os = translate(
            "splash", "Pyzo is open source software and freely available for
everyone."
        )
        text = splash_text.format(
            version=pyzo.__version__,
            text_title=text_title,
            text_version=text_version,
            text_os=text_os,
        )
        # Set text
        self._label.setText(text)
        layout = QtWidgets.QVBoxLayout(self)
        self.setLayout(layout)
        layout.addStretch(1)
        layout.addWidget(self._label, 0)
        layout.addStretch(1)
class SplashWidget(QtWidgets.QWidget):
    """A splash widget."""
    def __init__(self, parent, **kwargs):
        QtWidgets.QWidget.__init__(self, parent)
        self._left = LogoWidget(self)
        self._right = LabelWidget(self, **kwargs)
        # Layout
        layout = QtWidgets.QHBoxLayout(self)
        self.setLayout(layout)
        # layout.setContentsMargins(0,0,0,0)
        layout.setSpacing(25)
        layout.addStretch(1)
        layout.addWidget(self._left, 0)
        layout.addWidget(self._right, 0)
        layout.addStretch(1)
        # Change background of main window to create a splash-screen-efefct
        iconImage = "pyzologo256.png"
        iconImage = os.path.join(pyzo.pyzoDir, "resources", "appicons",
iconImage)
        iconImage = iconImage.replace(os.path.sep, "/")  # Fix for Windows
        self.setStyleSheet(STYLESHEET % iconImage)
if __name__ == "__main__":
    w = SplashWidget(None, distro="some arbitrary distro")
    w.resize(800, 600)
    w.show()
```

```
""" Module statusbar
Functionality for status bar in pyzo.
"""
from pyzo.qt import QtWidgets
class StatusBar(QtWidgets.QStatusBar):
    """
    Add a statusbar to main window
    """
    def __init__(self, parent=None):
        super().__init__(parent)
        # File encoding
        self.file_encoding = QtWidgets.QLabel(self)
        self.file_encoding.setFixedWidth(100)
        self.insertPermanentWidget(0, self.file_encoding, 0)
        # Cursor position
        self.cursor_pos = QtWidgets.QLabel(self)
        self.cursor_pos.setFixedWidth(190)
        self.insertPermanentWidget(1, self.cursor_pos, 0)
    def updateCursorInfo(self, editor):
        # Get current line number
        nrow = 0
        ncol = 0
        if editor:
            nrow = editor.textCursor().blockNumber()
            nrow += 1  # is ln as in line number area
            ncol = editor.textCursor().positionInBlock()
            ncol += 1
        position_txt = "Line: {}, Column: {} ".format(str(nrow), str(ncol))
        self.cursor_pos.setText(position_txt)
    def updateFileEncodingInfo(self, editor):
        fe_txt = ""
        if editor:
            fe_txt = editor.encoding.upper()
        self.file_encoding.setText(fe_txt)
```

```
import os
from pyzo.qt import QtCore, QtGui, QtWidgets
import pyzo
from pyzo.util import zon as ssdf
from pyzo.core.pyzoLogging import print  # noqa
import pyzo.core.baseTextCtrl
from pyzo.codeeditor.style import StyleFormat
SAMPLE = """
## Foo class
# This is a comment
class Foo:
''' This class does nothing. '''
    #TODO: be amazing
        def baz(self, arg1):
            return max(arg1, 42)
    bar = "Hello wor
""" + chr(
    160
)
class FakeEditor(pyzo.core.baseTextCtrl.BaseTextCtrl):
    """This "fake" editor emits a signal when
    the user clicks on a word with a token:
    a click on the word "class" emits with arg "syntax.keyword".
    It may be improved by adding text with specific token
    like Editor.text which are not present by default
    """
    tokenClicked = QtCore.Signal(str)
    def __init__(self, text=""):
        super().__init__()
        # set parser to enable syntaxic coloration
        self.setParser("python3")
        self.setReadOnly(False)
        self.setLongLineIndicatorPosition(30)
        self.setPlainText(SAMPLE)
    def mousePressEvent(self, event):
        super().mousePressEvent(event)
        # get the text position of the click
        pos = self.textCursor().columnNumber()
        tokens = self.textCursor().block().userData().tokens
        # Find the token which contains the click pos
        for tok in tokens:
            if tok.start <= pos <= tok.end:
                self.tokenClicked.emit(tok.description.key)
                break
class TitledWidget(QtWidgets.QWidget):
    """A litle helper class to "name" a widget :
    it displays a QLabel to left of the given widget"""
    def __init__(self, name, other):
        super().__init__()
        self.widget = other
        layout = QtWidgets.QHBoxLayout()
        layout.addWidget(QtWidgets.QLabel(text=name.capitalize().strip() + "
:"))
        layout.addWidget(other)
        self.setLayout(layout)
    def setFocus(self, val):
        self.widget.setFocus(val)
class ColorLineEdit(QtWidgets.QLineEdit):
    """A subclass of the QLineEdit that can open
    a QColorDialog on click of a button
    """
    def __init__(self, name, *args, **kwargs):
        """The name is displayed in the QColorDialog"""
        super().__init__(*args, **kwargs)
        self.name = name
        self.button = QtWidgets.QToolButton(self)
        self.button.setIcon(QtGui.QIcon(pyzo.icons.cog))
        self.button.setStyleSheet("border: 0px; padding: 0px")
        self.button.clicked.connect(self.openColorDialog)
        frameWidth =
self.style().pixelMetric(QtWidgets.QStyle.PM_DefaultFrameWidth)
        buttonSize = self.button.sizeHint()
        self.setStyleSheet(
            "QLineEdit {padding-right: %dpx; }" % (buttonSize.width() +
frameWidth + 1)
        )
        # self.setMinimumSize(max(100, buttonSize.width() + frameWidth*2 + 2),
        #                     max(self.minimumSizeHint().height(),
buttonSize.height() + frameWidth*2 + 2))
    def openColorDialog(self):
        """A simple function that opens a QColorDialog
        and link the dialog current color selection
        to the QLineEdit text
        """
        dlg = QtWidgets.QColorDialog(self)
        dlg.setWindowTitle("Pick a color for the " + self.name.lower())
        dlg.setCurrentColor(QtGui.QColor(self.text()))
        dlg.currentColorChanged.connect(lambda clr: self.setText(clr.name()))
        dlg.setModal(False)
        dlg.exec_()
    def resizeEvent(self, event):
        buttonSize = self.button.sizeHint()
        frameWidth =
self.style().pixelMetric(QtWidgets.QStyle.PM_DefaultFrameWidth)
        self.button.move(
            int(self.rect().right() - frameWidth - buttonSize.width()),
            int(self.rect().bottom() - buttonSize.height() + 1) // 2,
        )
        super().resizeEvent(event)
class StyleEdit(QtWidgets.QWidget):
    """The StyleLineEdit is a line that allows the edition
    of one style (i.e. "Editor.Text" or  "Syntax.identifier")
    with a given StyleElementDescription it find the editable
    parts and display the adaptated widgets for edition
    (checkbok for bold and italic, combo box for linestyles...).
    """
    styleChanged = QtCore.Signal(str, str)
    def __init__(self, defaultStyle, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # The styleKey is sent with the styleChanged signal for easy
identification
        self.styleKey = defaultStyle.key
        self.layout = layout = QtWidgets.QHBoxLayout()
        # The setters are used when setting the style
        self.setters = {}
        # TODO: the use of StyleFormat._parts should be avoided
        # We use the StyleFormat._parts keys, to find the elements
        # Useful to edits, because the property may return a value
        # Even if they were not defined in the defaultFormat
        fmtParts = defaultStyle.defaultFormat._parts
        # Add the widgets corresponding to the fields
        if "fore" in fmtParts:
            self.__add_clrLineEdit("fore", "Foreground")
        if "back" in fmtParts:
            self.__add_clrLineEdit("back", "Background")
        if "bold" in fmtParts:
            self.__add_checkBox("bold", "Bold")
        if "italic" in fmtParts:
            self.__add_checkBox("italic", "Italic")
        if "underline" in fmtParts:
            self.__add_comboBox(
                "underline", "Underline", "No", "Dotted", "Wave", "Full", "Yes"
            )
        if "linestyle" in fmtParts:
            self.__add_comboBox("linestyle", "Linestyle", "Dashed", "Dotted",
"Full")
        self.setLayout(layout)
        self.setSizePolicy(QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.Minimum)
    def __add_clrLineEdit(self, key, name):
        """this is a helper method to create a ColorLineEdit
        it adds the created widget (as a TitledWidget) to the layout and
        register a setter and listen to changes
        """
        clrEdit = ColorLineEdit(name)
        clrEdit.textChanged.connect(lambda txt, key=key: self.__update(key,
txt))
        self.setters[key] = clrEdit.setText
        self.layout.addWidget(TitledWidget(name, clrEdit), 0)
    def __add_checkBox(self, key, name):
        """this is a helper method to create a QCheckBox
        it adds the created widget (as a TitledWidget) to the layout and
        register a setter and listen to changes
        """
        checkBox = QtWidgets.QCheckBox()
        self.setters[key] = lambda val, check=checkBox: check.setCheckState(
            [QtCore.Qt.CheckState.Unchecked, QtCore.Qt.CheckState.Checked][val
== "yes"]
        )
        checkBox.stateChanged.connect(
            lambda state, key=key: self.__update(key, "yes" if state else "no")
        )
        self.layout.addWidget(TitledWidget(name, checkBox))
    def __add_comboBox(self, key, name, *items):
        """this is a helper method to create a comboBox
        it adds the created widget (as a TitledWidget) to the layout and
        register a setter and listen to changes
        """
        combo = QtWidgets.QComboBox()
        combo.addItems(items)
        combo.currentTextChanged.connect(lambda txt, key=key: self.__update(key,
txt))
```

```
        # Note: those setters may become problematic if
        # someone use the synonyms (defined in codeeditor/style.py)
        # i.e. a stylement is of form "linestyle:dashline"
        # instead of the "linestyle:dashed"
        self.setters[key] = lambda txt, cmb=combo:
cmb.setCurrentText(txt.capitalize())
        self.layout.addWidget(TitledWidget(name, combo))
    def __update(self, key, value):
        """this function is called everytime one of the children
        widget data has been modified by the user"""
        self.styleChanged.emit(self.styleKey, key + ":" + value)
    def setStyle(self, text):
        """updates every children to match the StyleFormat(text) fields"""
        style = StyleFormat(text)
        for key, setter in self.setters.items():
            setter(style[key])
    def setFocus(self, val):
        self.layout.itemAt(0).widget().setFocus(True)
class ThemeEditorWidget(QtWidgets.QWidget):
    """The ThemeEditorWidgets allows to edits themes,
    it has one StyleEdit widget per StyleElements ("Editor.Text",
    "Syntax.string"). It emits a signal on each style changes
    It also manages basic theme I/O :
        - adding new theme
        - renaming theme
    """
    styleChanged = QtCore.Signal(dict)
    done = QtCore.Signal(int)
    def __init__(self, themes, *args, editor=None, **kwargs):
        super().__init__(*args, **kwargs)
        # dict of themes, a deep copy of pyzo.themes
        self.themes = themes
        # We store the key name separate so we can easier track renames
        self.cur_theme_key = ""
        # The current theme being changed
        self.cur_theme = None
        # If an editor is given, connect to it
        self.editor = editor
        if self.editor is not None:
            self.editor.tokenClicked.connect(self.focusOnStyle)
            self.styleChanged.connect(self.editor.setStyle)
        # Display editables style formats in a scroll area
        self.scrollArea = scrollArea = QtWidgets.QScrollArea()
        self.scrollArea.setWidgetResizable(True)
        formLayout = QtWidgets.QFormLayout()
        self.styleEdits = {}
        # Add one pair of label and StyleEdit per style element description
        # to the formLayout and connect the StyleEdit signals to the
updatedStyle method
        for styleDesc in
pyzo.codeeditor.CodeEditor.getStyleElementDescriptions():
            label = QtWidgets.QLabel(text=styleDesc.name,
toolTip=styleDesc.description)
            label.setWordWrap(True)
            styleEdit = StyleEdit(styleDesc, toolTip=styleDesc.description)
            styleEdit.styleChanged.connect(self.updatedStyle)
            self.styleEdits[styleDesc.key] = styleEdit
            formLayout.addRow(label, styleEdit)
        wrapper = QtWidgets.QWidget()
        wrapper.setLayout(formLayout)
        wrapper.setMinimumWidth(650)
        scrollArea.setWidget(wrapper)
        # Basic theme I/O
        curThemeLbl = QtWidgets.QLabel(text="Themes :")
        self.curThemeCmb = curThemeCmb = QtWidgets.QComboBox()
        current_index = -1
        for i, themeName in enumerate(self.themes.keys()):
            # We store the themeName in data in case the user renames one
            curThemeCmb.addItem(themeName, userData=themeName)
            if themeName == pyzo.config.settings.theme.lower():
                current_index = i
        curThemeCmb.addItem("New...")
        loadLayout = QtWidgets.QHBoxLayout()
        loadLayout.addWidget(curThemeLbl)
        loadLayout.addWidget(curThemeCmb)
        self.saveBtn = saveBtn = QtWidgets.QPushButton(text="Save")
        saveBtn.clicked.connect(self.saveTheme)
        exitBtn = QtWidgets.QPushButton(text="Apply theme")
        exitBtn.clicked.connect(self.ok)
        exitLayout = QtWidgets.QHBoxLayout()
        exitLayout.addWidget(exitBtn)
        exitLayout.addWidget(saveBtn)
        # Packing it up
        mainLayout = QtWidgets.QVBoxLayout()
        mainLayout.addLayout(loadLayout)
        mainLayout.addWidget(scrollArea)
        mainLayout.addLayout(exitLayout)
        self.setLayout(mainLayout)
        curThemeCmb.currentIndexChanged.connect(self.indexChanged)
        curThemeCmb.currentTextChanged.connect(self.setTheme)
        # Init
        if current_index >= 0:
            curThemeCmb.setCurrentIndex(current_index)
            self.setTheme(pyzo.config.settings.theme)
    def createTheme(self):
        """Create a new theme based on the current
        theme selected.
        """
        index = self.curThemeCmb.currentIndex()
        if index != self.curThemeCmb.count() - 1:
            return self.curThemeCmb.setCurrentIndex(self.curThemeCmb.count() -
1)
        # Select a new name
        t = "new_theme_x"
        i = 1
        themeName = t.replace("x", str(i))
        while themeName in self.themes:
            i += 1
            themeName = t.replace("x", str(i))
        # Create new theme
        new_theme = {"name": themeName, "data": {}, "builtin": False}
        if self.cur_theme:
            new_theme["data"] = self.cur_theme["data"].copy()
        self.cur_theme_key = themeName
        self.cur_theme = new_theme
        self.themes[themeName] = new_theme
        self.curThemeCmb.setItemText(index, themeName)
        self.curThemeCmb.setItemData(index, themeName)
        self.curThemeCmb.setEditable(True)
        self.curThemeCmb.lineEdit().setCursorPosition(0)
        self.curThemeCmb.lineEdit().selectAll()
        self.saveBtn.setEnabled(True)
        self.curThemeCmb.addItem(
            "New...",
        )
    def setTheme(self, name):
        """Set the theme by its name. The combobox becomes editable only
        if the theme is not builtin. This method is connected to the signal
        self.curThemeCmb.currentTextChanged ; so it also filters
        parasites events"""
        name = name.lower()
        if name != self.curThemeCmb.currentText():
            # An item was added to the comboBox
            # But it's not a user action so we quit
            print(" -> Cancelled because this was not a user action")
            return
        if self.cur_theme_key == self.curThemeCmb.currentData():
            # The user renamed an existing theme
            self.cur_theme["name"] = name
            return
        if name not in self.themes:
            return
        # Sets the curent theme key
        self.cur_theme_key = name
        self.cur_theme = self.themes[name]
        if self.cur_theme["builtin"]:
            self.saveBtn.setEnabled(False)
            self.saveBtn.setText("Cannot save builtin style")
        else:
            self.saveBtn.setEnabled(True)
            self.saveBtn.setText("Save")
        self.curThemeCmb.setEditable(not self.cur_theme["builtin"])
        for key, le in self.styleEdits.items():
            if key in self.cur_theme["data"]:
                try:
                    le.setStyle(self.cur_theme["data"][key])
                except Exception as e:
                    print(
                        "Exception while setting style", key, "for theme", name,
":", e
                    )
    def saveTheme(self):
        """Saves the current theme to the disk, in appDataDir/themes"""
        if self.cur_theme["builtin"]:
            return
        themeName = self.curThemeCmb.currentText().strip()
        if not themeName:
            return
        # Get user theme dir and make sure it exists
        dir = os.path.join(pyzo.appDataDir, "themes")
        os.makedirs(dir, exist_ok=True)
        # Try to delete the old file if it exists (useful if it was renamed)
        try:
            os.remove(os.path.join(dir, self.cur_theme_key + ".theme"))
        except Exception:
            pass
        # This is the needed because of the SSDF format:
        # it doesn't accept dots, so we put underscore instead
        data = {x.replace(".", "_"): y for x, y in
self.cur_theme["data"].items()}
        fname = os.path.join(dir, themeName + ".theme")
        ssdf.save(fname, {"name": themeName, "data": data})
        print("Saved theme '%s' to '%s'" % (themeName, fname))
    def ok(self):
        """On user click saves the cur_theme if modified
        and restart pyzo if the theme changed"""
        prev = pyzo.config.settings.theme
        new = self.cur_theme["name"]
        self.saveTheme()
        if prev != new:
            pyzo.config.settings.theme = new
            pyzo.saveConfig()
            # This may be better
            pyzo.main.restart()
        else:
            self.done.emit(1)
    def indexChanged(self, index):
        # User selected the "new..." button
        if index == self.curThemeCmb.count() - 1:
            self.createTheme()
    def focusOnStyle(self, key):
        self.styleEdits[key].setFocus(True)
        self.scrollArea.ensureWidgetVisible(self.styleEdits[key])
    def updatedStyle(self, style, text):
        fmt = StyleFormat(self.cur_theme["data"][style])
        fmt.update(text)
        self.cur_theme["data"][style] = str(fmt)
        self.styleChanged.emit({style: text})
class EditColorDialog(QtWidgets.QDialog):
    """This dialog allows to edit color schemes,
    it is composed of two main components :
        - a "fake" editor to visualize the changes
        - a theme editor to make the edits
    """
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.setWindowTitle("Color scheme")
        size = 1200, 800
        offset = 0
        size2 = size[0], size[1] + offset
        self.resize(*size2)
        # Make a deep copy
        themes = {}
        for name, theme in pyzo.themes.items():
            theme = theme.copy()
            theme["data"] = theme["data"].copy()
            themes[name] = theme
        self.editor = FakeEditor()
        self.editColor = ThemeEditorWidget(themes=themes, editor=self.editor)
        self.editColor.done.connect(self.done)
        layout = QtWidgets.QHBoxLayout()
        layout.addWidget(self.editor, 1)
        layout.addWidget(self.editColor, 2)
        self.setLayout(layout)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Package core - the core of Pyzo.
"""
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
import os
import sys
import time
import bdb
import traceback
class Debugger(bdb.Bdb):
    """Debugger for the pyzo kernel, based on bdb."""
    def __init__(self):
        self._wait_for_mainpyfile = False  # from pdb, do we need this?
        bdb.Bdb.__init__(self)
        self._debugmode = 0  # 0: no debug,  1: postmortem,  2: full debug
        self._files_with_offset = []
    def clear_all_breaks(self):
        bdb.Bdb.clear_all_breaks(self)
        self._files_with_offset = []
    def trace_dispatch(self, frame, event, arg):
        # Overload to deal with offset in filenames
        # (cells or lines being executed)
        ori_filename = frame.f_code.co_filename
        if "+" in ori_filename and ori_filename not in self._files_with_offset:
            clean_filename, offset = ori_filename.rsplit("+", 1)
            try:
                offset = int(offset)
            except Exception:
                offset = None
            if offset is not None:
                # This is a cell or selected lines being executed
                self._files_with_offset.append(ori_filename)
                if clean_filename.startswith("<"):
                    self.fncache[ori_filename] = ori_filename
                for i in self.breaks.get(clean_filename, []):
                    self.set_break(ori_filename, i - offset)
        return bdb.Bdb.trace_dispatch(self, frame, event, arg)
    def interaction(self, frame, traceback=None, pm=False):
        """Enter an interaction-loop for debugging. No GUI events are
        processed here. We leave this event loop at some point, after
        which the conrol flow will proceed.
        This is called to enter debug-mode at a breakpoint, or to enter
```

```
post-mortem debugging.
"""
interpreter = sys._pyzoInterpreter
# Collect frames
frames = []
while frame:
    if frame is self.botframe:
        break
    co_filename = frame.f_code.co_filename
    if "pyzokernel" in co_filename:
        break  # pyzo kernel
    if "interactiveshell.py" in co_filename:
        break  # IPython kernel
    frames.insert(0, frame)
    frame = frame.f_back
# Tell interpreter our stack
if frames:
    interpreter._dbFrames = frames
    interpreter._dbFrameIndex = len(interpreter._dbFrames)
    frame = interpreter._dbFrames[interpreter._dbFrameIndex - 1]
    interpreter._dbFrameName = frame.f_code.co_name
    interpreter.locals = frame.f_locals
    interpreter.globals = frame.f_globals
# Let the IDE know (
# "self._debugmode = 1 if pm else 2" does not work not on py2.4)
if pm:
    self._debugmode = 1
else:
    self._debugmode = 2
self.writestatus()
# Enter interact loop. We may hang in here for a while ...
self._interacting = True
while self._interacting:
    time.sleep(0.05)
    interpreter.process_commands()
    pe = os.getenv("PYZO_PROCESS_EVENTS_WHILE_DEBUGGING", "").lower()
    if pe in ("1", "true", "yes"):
        interpreter.guiApp.process_events()
# Reset
self._debugmode = 0
interpreter.locals = interpreter._main_locals
interpreter.globals = None
interpreter._dbFrames = []
```

```
        self.writestatus()
    def stopinteraction(self):
        """Stop the interaction loop."""
        self._interacting = False
    def set_on(self):
        """To turn debugging on right before executing code."""
        # Reset and set bottom frame
        self.reset()
        self.botframe = sys._getframe().f_back
        # Don't stop except at breakpoints or when finished
        # We do: self._set_stopinfo(self.botframe, None, -1) from set_continue
        # But write it all out because py2.4 does not have _set_stopinfo
        self.stopframe = self.botframe
        self.returnframe = None
        self.quitting = False
        self.stoplineno = -1
        # Set tracing or not
        if self.breaks:
            sys.settrace(self.trace_dispatch)
        else:
            sys.settrace(None)
    def message(self, msg):
        """Alias for interpreter.write(), but appends a newline.
        Writes to stderr.
        """
        sys._pyzoInterpreter.write(msg + "\n")
    def error(self, msg):
        """method used in some code that we copied from pdb."""
        raise self.message("*** " + msg)
    def writestatus(self):
        """Write the debug status so the IDE can take action."""
        interpreter = sys._pyzoInterpreter
        # Collect frames info
        frames = []
        for f in interpreter._dbFrames:
            # Get fname and lineno, and correct if required
            fname, lineno = f.f_code.co_filename, f.f_lineno
            fname, lineno = interpreter.correctfilenameandlineno(fname, lineno)
            if not fname.startswith("<"):
                fname2 = os.path.abspath(fname)
                if os.path.isfile(fname2):
                    fname = fname2
            frames.append((fname, lineno, f.f_code.co_name))
            # Build string
            # text = 'File "%s", line %i, in %s' % (
            #                         fname, lineno, f.f_code.co_name)
            # frames.append(text)
        # Send info object
        state = {
            "index": interpreter._dbFrameIndex,
            "frames": frames,
            "debugmode": self._debugmode,
        }
        interpreter.context._stat_debug.send(state)
    ## Stuff that we need to overload
    # Overload set_break to also allow non-existing filenames like "<tmp 1"
    def set_break(self, filename, lineno, temporary=False, cond=None,
funcname=None):
        filename = self.canonic(filename)
        list = self.breaks.setdefault(filename, [])
        if lineno not in list:
            list.append(lineno)
        bdb.Breakpoint(filename, lineno, temporary, cond, funcname)
    # Prevent stopping in bdb code or pyzokernel code
    def stop_here(self, frame):
        result = bdb.Bdb.stop_here(self, frame)
        if result:
            return ("bdb.py" not in frame.f_code.co_filename) and (
                "pyzokernel" not in frame.f_code.co_filename
            )
    def do_clear(self, arg):
        """ """
        # Clear breakpoints, we need to overload from Bdb,
        # but do not expose this command to the user.
        """cl(ear) filename:lineno\ncl(ear) [bpnumber [bpnumber...]]
        With a space separated list of breakpoint numbers, clear
        those breakpoints.  Without argument, clear all breaks (but
        first ask confirmation).  With a filename:lineno argument,
        clear all breaks at that line in that file.
        """
        if not arg:
            bplist = [bp for bp in bdb.Breakpoint.bpbynumber if bp]
            self.clear_all_breaks()
            for bp in bplist:
                self.message("Deleted %s" % bp)
            return
        if ":" in arg:
            # Make sure it works for "clear C:\foo\bar.py:12"
            i = arg.rfind(":")
            filename = arg[:i]
            arg = arg[i + 1 :]
            try:
                lineno = int(arg)
            except ValueError:
                err = "Invalid line number (%s)" % arg
            else:
                bplist = self.get_breaks(filename, lineno)
                err = self.clear_break(filename, lineno)
            if err:
                self.error(err)
            else:
                for bp in bplist:
                    self.message("Deleted %s" % bp)
            return
        numberlist = arg.split()
        for i in numberlist:
            try:
                bp = self.get_bpbynumber(i)
            except ValueError:
                self.error("Cannot get breakpoint by number.")
            else:
                self.clear_bpbynumber(i)
                self.message("Deleted %s" % bp)
    def user_call(self, frame, argument_list):
        """This method is called when there is the remote possibility
        that we ever need to stop in this function."""
        if self._wait_for_mainpyfile:
            return
        if self.stop_here(frame):
            self.message("--Call--")
            self.interaction(frame, None)
    def user_line(self, frame):
        """This function is called when we stop or break at this line."""
        if self._wait_for_mainpyfile:
            if (
                self.mainpyfile != self.canonic(frame.f_code.co_filename)
                or frame.f_lineno <= 0
            ):
                return
            self._wait_for_mainpyfile = False
        if True:  # self.bp_commands(frame):  from pdb
            self.interaction(frame, None)
    def user_return(self, frame, return_value):
        """This function is called when a return trap is set here."""
        if self._wait_for_mainpyfile:
            return
        frame.f_locals["__return__"] = return_value
        self.message("--Return--")
        self.interaction(frame, None)
    def user_exception(self, frame, exc_info):
        """This function is called if an exception occurs,
        but only if we are to stop at or just below this level."""
        if self._wait_for_mainpyfile:
            return
        exc_type, exc_value, exc_traceback = exc_info
        frame.f_locals["__exception__"] = exc_type, exc_value
        self.message(traceback.format_exception_only(exc_type,
exc_value)[-1].strip())
        self.interaction(frame, exc_traceback)
    ## Commands
    def do_help(self, arg):
        """Get help on debug commands."""
        # Collect docstrings
        docs = {}
        for name in dir(self):
            if name.startswith("do_"):
                doc = getattr(self, name).__doc__
                if doc:
                    docs[name[3:]] = doc.strip()
        if not arg:
            print("All debug commands:")
            # Show docs in  order
            for name in [
                "start",
                "stop",
                "frame",
                "up",
                "down",
                "next",
                "step",
                "return",
                "continue",
                "where",
                "events",
            ]:
                doc = docs.pop(name)
                name = name.rjust(10)
                print(" %s - %s" % (name, doc))
            # Show rest
            for name in docs:
                doc = docs[name]
                name = name.rjust(10)
                print(" %s - %s" % (name, doc))
        else:
            # Show specific doc
            name = arg.lower()
            doc = docs.get(name, None)
            if doc is not None:
                print("%s - %s" % (name, doc))
            else:
                print("Unknown debug command: %s" % name)
    def do_start(self, arg):
        """Start postmortem debugging from the last uncaught exception."""
        # Get traceback
        try:
            tb = sys.last_traceback
        except AttributeError:
            tb = None
        # Get top frame
        frame = None
        while tb:
            frame = tb.tb_frame
            tb = tb.tb_next
        # Interact, or not
        if self._debugmode:
            self.message("Already in debug mode.")
        elif frame:
            self.interaction(frame, None, pm=True)
        else:
            self.message("No debug information available.")
    def do_frame(self, arg):
        """Go to the i'th frame in the stack."""
        interpreter = sys._pyzoInterpreter
        if not self._debugmode:
            self.message("Not in debug mode.")
        else:
            # Set frame index
            interpreter._dbFrameIndex = int(arg)
            if interpreter._dbFrameIndex < 1:
                interpreter._dbFrameIndex = 1
            elif interpreter._dbFrameIndex > len(interpreter._dbFrames):
                interpreter._dbFrameIndex = len(interpreter._dbFrames)
            # Set name and locals
            frame = interpreter._dbFrames[interpreter._dbFrameIndex - 1]
            interpreter._dbFrameName = frame.f_code.co_name
            interpreter.locals = frame.f_locals
            interpreter.globals = frame.f_globals
            self.writestatus()
    def do_up(self, arg):
        """Go one frame up the stack."""
        interpreter = sys._pyzoInterpreter
        if not self._debugmode:
            self.message("Not in debug mode.")
        else:
            # Decrease frame index
            interpreter._dbFrameIndex -= 1
            if interpreter._dbFrameIndex < 1:
                interpreter._dbFrameIndex = 1
            # Set name and locals
            frame = interpreter._dbFrames[interpreter._dbFrameIndex - 1]
            interpreter._dbFrameName = frame.f_code.co_name
            interpreter.locals = frame.f_locals
            interpreter.globals = frame.f_globals
            self.writestatus()
    def do_down(self, arg):
        """Go one frame down the stack."""
        interpreter = sys._pyzoInterpreter
        if not self._debugmode:
            self.message("Not in debug mode.")
        else:
            # Increase frame index
            interpreter._dbFrameIndex += 1
            if interpreter._dbFrameIndex > len(interpreter._dbFrames):
                interpreter._dbFrameIndex = len(interpreter._dbFrames)
            # Set name and locals
            frame = interpreter._dbFrames[interpreter._dbFrameIndex - 1]
            interpreter._dbFrameName = frame.f_code.co_name
            interpreter.locals = frame.f_locals
            interpreter.globals = frame.f_globals
            self.writestatus()
    def do_stop(self, arg):
        """Stop debugging, terminate process execution."""
        # Can be done both in postmortem and normal debugging
        if not self._debugmode:
            self.message("Not in debug mode.")
        else:
            self.set_quit()
            self.stopinteraction()
    def do_where(self, arg):
        """Print the stack trace and indicate the current frame."""
        interpreter = sys._pyzoInterpreter
        if not self._debugmode:
            self.message("Not in debug mode.")
        else:
            lines = []
            for i in range(len(interpreter._dbFrames)):
                frameIndex = i + 1
                f = interpreter._dbFrames[i]
                # Get fname and lineno, and correct if required
                fname, lineno = f.f_code.co_filename, f.f_lineno
                fname, lineno = interpreter.correctfilenameandlineno(fname,
lineno)
                # Build string
                text = 'File "%s", line %i, in %s' % (fname, lineno,
f.f_code.co_name)
                if frameIndex == interpreter._dbFrameIndex:
                    lines.append("-> %i: %s" % (frameIndex, text))
                else:
                    lines.append("   %i: %s" % (frameIndex, text))
            lines.append("")
            sys.stdout.write("\n".join(lines))
    def do_continue(self, arg):
        """Continue the program execution."""
        if self._debugmode == 0:
            self.message("Not in debug mode.")
        elif self._debugmode == 1:
            self.message("Cannot use 'continue' in postmortem debug mode.")
        else:
            self.set_continue()
            self.stopinteraction()
    def do_step(self, arg):
        """Execute the current line, stop ASAP (step into)."""
        if self._debugmode == 0:
            self.message("Not in debug mode.")
        elif self._debugmode == 1:
            self.message("Cannot use 'step' in postmortem debug mode.")
        else:
            self.set_step()
            self.stopinteraction()
    def do_next(self, arg):
        """Continue execution until the next line (step over)."""
        interpreter = sys._pyzoInterpreter
        if self._debugmode == 0:
            self.message("Not in debug mode.")
        elif self._debugmode == 1:
            self.message("Cannot use 'next' in postmortem debug mode.")
        else:
            frame = interpreter._dbFrames[-1]
            self.set_next(frame)
            self.stopinteraction()
    def do_return(self, arg):
        """Continue execution until the current function returns (step out)."""
        interpreter = sys._pyzoInterpreter
        if self._debugmode == 0:
            self.message("Not in debug mode.")
        elif self._debugmode == 1:
            self.message("Cannot use 'return' in postmortem debug mode.")
        else:
            frame = interpreter._dbFrames[-1]
            self.set_return(frame)
            self.stopinteraction()
    def do_events(self, arg):
        """Process GUI events for the integrated GUI toolkit."""
        interpreter = sys._pyzoInterpreter
        interpreter.guiApp.process_events()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
"""
Module to integrate GUI event loops in the Pyzo interpreter.
This specifies classes that all have the same interface. Each class
wraps one GUI toolkit.
Support for PyQt4, WxPython, FLTK, GTK, TK.
"""
import sys
import time
from pyzokernel import printDirect
# Warning message.
mainloopWarning = (
    """
Note: The GUI event loop is already running in the pyzo kernel. Be aware
that the function to enter the main loop does not block.
""".strip()
    + "\n"
)
# Qt has its own message
mainloopWarning_qt = """
Note on using QApplication.exec_():
The GUI event loop is already running in the pyzo kernel, and exec_()
does not block. In most cases your app should run fine without the need
for modifications. For clarity, this is what the pyzo kernel does:
- Prevent deletion of objects in the local scope of functions leading to exec_()
- Prevent system exit right after the exec_() call
""".lstrip()
# Print the main loop warning at most once
_printed_warning = False
def print_mainloop_warning(msg=None):
    global _printed_warning
    if not _printed_warning:
        _printed_warning = True
        msg = msg or mainloopWarning
        printDirect(msg)
class App_base:
    """Defines the interface."""
    def process_events(self):
        pass
    def _keyboard_interrupt(self, signum=None, frame=None):
        interpreter = sys._pyzoInterpreter
        interpreter.write("\nKeyboardInterrupt\n")
        interpreter._resetbuffer()
        if interpreter.more:
            interpreter.more = 0
            interpreter.newPrompt = True
    def run(self, repl_callback, sleep_time=0.01):
        """Very simple mainloop. Subclasses can overload this to use
        the native event loop. Attempt to process GUI events every so often.
        """
        # The sleep_time is 100Hz by default, which seems like an ok balance
        # between CPU strain and smooth animations. Ideally we'd run a
        # real event loop though, that is fast when needed and just sleeps
        # when the gui is idle, saving battery life.
        if hasattr(time, "perf_counter"):
            perf_counter = time.perf_counter
        else:
            perf_counter = time.time
        perf_counter
        repl_time = 0.099
        next_repl = perf_counter() + repl_time
        # The toplevel while-loop is just to catch Keyboard interrupts
        # and then proceed. The inner while-loop is the actual event loop.
        while True:
            try:
                while True:
                    time.sleep(sleep_time)
                    self.process_events()
                    if perf_counter() > next_repl:
                        next_repl = perf_counter() + repl_time
                        repl_callback()
            except KeyboardInterrupt:
                self._keyboard_interrupt()
            except TypeError:
                # For some reason, when wx is integrated, keyboard interrupts
                # result in a TypeError.
                # I tried to find the source, but did not find it. If anyone
                # has an idea, please e-mail me!
                if "_wx" in self.__class__.__name__.lower():
                    self._keyboard_interrupt()
    def quit(self):
        raise SystemExit()
```

```
class App_nogui(App_base):
    """The app for when there is no GUI."""
    def run(self, repl_callback):
        # Move at a slow pace - there is no gui to keep running
        super().run(repl_callback, 0.1)
# Experimental and WIP - not used at the moment
class App_asyncio_new(App_base):
    """Based on asyncio (standard Python) event loop.
    Actually run an event loop and support switching to another loop.
    """
    def __init__(self):
        import asyncio
        asyncio.integrate_with_ide = self.enable
        self._loop = None
    def run(self, repl_callback):
        import asyncio
        self._repl_callback = repl_callback
        self._sleep_time = 0.1
        loop = asyncio.get_event_loop_policy().get_event_loop()
        self.enable(loop, True)
    def enable(self, loop, run=False):
        if loop is not self._loop:
            self.swap_loops_when_new_one_starts(self._loop, loop)
            loop.call_later(self._sleep_time, self._ping_repl_callback)
        if run:
            loop.run_forever()
    def swap_loops_when_new_one_starts(self, old_loop, new_loop):
        def new_run_forever(*args, **kwargs):
            if old_loop and old_loop.is_running():
                old_loop.stop()
            self._loop = new_loop
            if not new_loop.is_running():
                new_loop.original_run_forever(*args, **kwargs)
        if not hasattr(new_loop, "original_run_forever"):
            new_loop.original_run_forever = new_loop.run_forever
        new_loop.run_forever = new_run_forever
    def _ping_repl_callback(self):
        import asyncio
        self._repl_callback()
        # Get loop that (probably) called this
        try:
            loop = asyncio.get_running_loop()
        except Exception:
            loop = None
        # If its the same as our current loop, we want to be called again
        if loop:
            self._loop.call_later(self._sleep_time, self._ping_repl_callback)
    def quit(self):
        if self._loop:
            self._loop.stop()
        # raise SystemExit()
class App_asyncio(App_base):
    """Based on asyncio (standard Python) event loop.
    We do not run the event loop and a timer to keep the REPL active, because
    asyncio does allow creating new loops while one is running. So we stick to
    a simple and non-intrusive mechanism to regularly process events from
    whatever event loop is current at any given moment.
    """
    def __init__(self):
        import asyncio
        self.app = asyncio.get_event_loop_policy().get_event_loop()
        self.app._in_event_loop = "Pyzo"
        self._warned_about_process_events = False
        self._blocking = False
        # Hijack
        # Prevent entering forever, not giving control back to repl
        self.app._original_run_forever = self.app.run_forever
        self.app.run_forever = self.stub_run_forever
        # The run_until_complete() calls run_forever and then checks that future
completed
        self.app._original_run_until_complete = self.app.run_until_complete
        self.app.run_until_complete = self.stub_run_until_complete
        # Prevent the loop from being destroyed
        self.app._original_close = self.app.close
        self.app.close = self.stub_close
        # Stop is fine, since we don't "run" the loop, but just keep it active
    def stub_run_until_complete(self, *args):
        self._blocking = True
        return self.app._original_run_until_complete(*args)
    def stub_run_forever(self):
        if self._blocking:
            self._blocking = False
            self.app._original_run_forever()
    def stub_close(self):
        pass
    def process_events(self):
        import asyncio
        try:
            loop = asyncio.get_event_loop_policy().get_event_loop()
        except Exception:
            loop = None
        if loop is not self.app:
            return  # The loop was replaced
        loop = self.app
        if loop.is_closed():
            pass  # not much we can do
        elif loop.is_running():
            if not self._warned_about_process_events:
                print("Warning: cannot process events synchronously in asyncio")
                self._warned_about_process_events = True
        else:
            # First calling stop and then run_forever() process all pending
events.
            # We do this multiple times to work around the limited frequence
that this
            # method gets called, but we need to use a private attribute for
that :/
            for i in range(20):
                loop.stop()
                loop._original_run_forever()
                if len(getattr(loop, "_ready", ())) == 0:
                    break
    def quit(self):
        loop = self.app
        try:
            if not loop.is_closed():
                loop.close()
        finally:
            raise SystemExit()
class App_tk(App_base):
    """Tries to import tkinter and returns a withdrawn tkinter root
    window.  If tkinter is already imported or not available, this
    returns None.
    Modifies tkinter's mainloop with a dummy so when a module calls
    mainloop, it does not block.
    """
    def __init__(self):
        # Try importing
        import sys
        if sys.version[0] == "3":
            import tkinter
        else:
            import Tkinter as tkinter
        # Replace mainloop. Note that a root object obtained with
        # tkinter.Tk() has a mainloop method, which will simply call
        # tkinter.mainloop().
        def dummy_mainloop(*args, **kwargs):
            print_mainloop_warning()
        tkinter.Misc.mainloop = dummy_mainloop
        tkinter.mainloop = dummy_mainloop
        # Create tk "main window" that has a Tcl interpreter.
        # Withdraw so it's not shown. This object can be used to
        # process events for any other windows.
        r = tkinter.Tk()
        r.withdraw()
        # Store the app instance to process events
        self.app = r
        # Notify that we integrated the event loop
        self.app._in_event_loop = "Pyzo"
        tkinter._in_event_loop = "Pyzo"
    def process_events(self):
        self.app.update()
class App_fltk(App_base):
    """Hijack fltk 1.
    This one is easy. Just call fl.wait(0.0) now and then.
    Note that both tk and fltk try to bind to PyOS_InputHook. Fltk
    will warn about not being able to and Tk does not, so we should
    just hijack (import) fltk first. The hook that they try to fetch
    is not required in pyzo, because the pyzo interpreter will keep
    all GUI backends updated when idle.
    """
    def __init__(self):
        # Try importing
        import fltk as fl
        import types
        # Replace mainloop with a dummy
        def dummyrun(*args, **kwargs):
            print_mainloop_warning()
        fl.Fl.run = types.MethodType(dummyrun, fl.Fl)
        # Store the app instance to process events
        self.app = fl.Fl
        # Notify that we integrated the event loop
        self.app._in_event_loop = "Pyzo"
        fl._in_event_loop = "Pyzo"
    def process_events(self):
        self.app.wait(0)
class App_fltk2(App_base):
    """Hijack fltk 2."""
    def __init__(self):
        # Try importing
        import fltk2 as fl
        # Replace mainloop with a dummy
        def dummyrun(*args, **kwargs):
            print_mainloop_warning()
        fl.run = dummyrun
        # Return the app instance to process events
        self.app = fl
        # Notify that we integrated the event loop
        self.app._in_event_loop = "Pyzo"
    def process_events(self):
        # is this right?
        self.app.wait(0)
class App_tornado(App_base):
    """Hijack Tornado event loop.
    Tornado does have a function to process events, but it does not
    work when the event loop is already running. Therefore we don't
    enter the real Tornado event loop, but just poll it regularly.
    """
    def __init__(self):
        # Try importing
        import tornado.ioloop
        # Get the "app" instance
        self.app = tornado.ioloop.IOLoop.instance()
        # Replace mainloop with a dummy
        def dummy_start():
            print_mainloop_warning()
            sys._pyzoInterpreter.ignore_sys_exit = True
            self.app.add_callback(reset_sys_exit)
        def dummy_stop():
            pass
        def reset_sys_exit():
            sys._pyzoInterpreter.ignore_sys_exit = False
        def run_sync(func, timeout=None):
            self.app.start = self.app._original_start
            try:
                self.app._original_run_sync(func, timeout)
            finally:
                self.app.start = self.app._dummy_start
        #
        self.app._original_start = self.app.start
        self.app._dummy_start = dummy_start
        self.app.start = self.app._dummy_start
        #
        self.app._original_stop = self.app.stop
        self.app._dummy_stop = dummy_stop
        self.app.stop = self.app._dummy_stop
        #
        self.app._original_run_sync = self.app.run_sync
        self.app.run_sync = run_sync
        # Notify that we integrated the event loop
        self.app._in_event_loop = "Pyzo"
        self._warned_about_process_events = False
    def process_events(self):
        if not self._warned_about_process_events:
            print("Warning: cannot process events synchronously in Tornado")
            self._warned_about_process_events = True
        # self.app.run_sync(lambda x=None: None)
    def run(self, repl_callback):
        from tornado.ioloop import PeriodicCallback
        # Create timer
        self._timer = PeriodicCallback(repl_callback, 0.05 * 1000)
        self._timer.start()
        # Enter mainloop
        self.app._original_start()
        while True:
            try:
                self.app._original_start()
            except KeyboardInterrupt:
                self._keyboard_interrupt()
                self.app._original_stop()
                continue
            break
    def quit(self):
        self.app._original_stop()
        # raise SystemExit()
class App_qt(App_base):
    """Common functionality for pyqt and pyside"""
    def __init__(self):
        import types
        # Try importing qt
        QtGui, QtCore = self.importCoreAndGui()
        self._QtGui, self._QtCore = QtGui, QtCore
        # Store the real application class
        if not hasattr(QtGui, "real_QApplication"):
            QtGui.real_QApplication = QtGui.QApplication
        class QApplication_hijacked(QtGui.QApplication):
            """QApplication_hijacked(*args, **kwargs)
            Hijacked QApplication class. This class has a __new__()
            method that always returns the global application
            instance, i.e. QtGui.qApp.
            The QtGui.qApp instance is an instance of the original
            QtGui.QApplication, but with its __init__() and exec_()
            methods replaced.
            You can subclass this class; the global application instance
            will be given the methods and attributes so it will behave
            like the subclass.
            """
            def __new__(cls, *args, **kwargs):
                # Get the singleton application instance
                theApp = cls.instance()
                # Instantiate an original QApplication instance if we need to
                if theApp is None:
                    theApp = QtGui.real_QApplication(*args, **kwargs)
                    QtGui.qApp = theApp
                for key in ['__init__', 'exec_', 'quit']:
                    if not hasattr(cls, key):
                        continue
                    # Skip attributes that we already have
                    val = getattr(cls, key)
                    if hasattr(theApp.__class__, key):
                        if hash(val) == hash(getattr(theApp.__class__, key)):
                            continue
                    # Make method?
                    if hasattr(val, "__call__"):
                        if hasattr(val, "im_func"):
                            val = val.im_func  # Python 2.x
                        val = types.MethodType(val, theApp.__class__)
                    # Set attribute on app instance (not the class!)
                    try:
                        setattr(theApp, key, val)
                    except Exception:
                        pass  # tough luck
                setattr(theApp, "exec", theApp.exec_)
                # Call init function (in case the user overloaded it)
                theApp.__init__(*args, **kwargs)
                # Return global app object (modified to the users needs)
                return theApp
            def __init__(self, *args, **kwargs):
                pass
            def exec_(self, *args, **kwargs):
                """This function does nothing, except printing a
                warning message. The point is that a Qt App can crash
                quite hard if an object goes out of scope, and the error
                is not obvious.
                """
                print_mainloop_warning(mainloopWarning_qt)
                # Store local namespaces (scopes) of any functions that
                # precede this call. It might have a widget or application
                # object that should not be deleted ...
                import inspect, __main__
                for caller in inspect.stack()[1:]:
                    frame, name = caller[0], caller[3]
                    if name.startswith("<"):  # most probably "<module>"
                        break
                    else:
                        __main__.__dict__[name + "_locals"] = frame.f_locals
                # Tell interpreter to ignore any system exits
                sys._pyzoInterpreter.ignore_sys_exit = True
                # But re-enable it as soon as *this event* is processed
                def reEnableSysExit():
                    sys._pyzoInterpreter.ignore_sys_exit = False
                self._reEnableSysExitTimer = timer = QtCore.QTimer()
                timer.singleShot(0, reEnableSysExit)
            def quit(self, *args, **kwargs):
                """Do not quit if Qt app quits."""
                pass
        # Instantiate application object
        self.app = QApplication_hijacked([""])
        # Keep it alive even if all windows are closed
        self.app.setQuitOnLastWindowClosed(False)
        # Replace app class
        QtGui.QApplication = QApplication_hijacked
        # Notify that we integrated the event loop
        self.app._in_event_loop = "Pyzo"
        QtGui._in_event_loop = "Pyzo"
        # Use sys.excepthook to catch keyboard interrupts that occur
        # in event handlers. We also want to call the curren hook
        self._original_excepthook = sys.excepthook
        sys.excepthook = self._excepthook
    def _excepthook(self, type, value, traceback):
        if issubclass(type, KeyboardInterrupt):
            self._keyboard_interrupt()
        elif self._original_excepthook is not None:
            return self._original_excepthook(type, value, traceback)
    def process_events(self):
        self.app.sendPostedEvents()
        self.app.processEvents()
    def run(self, repl_callback):
        # Create timer
        timer = self._timer = self._QtCore.QTimer()
        timer.setSingleShot(False)
        timer.setInterval(int(0.1 * 1000))  # ms
        timer.timeout.connect(repl_callback)
        timer.start()
        # Enter Qt mainloop
        # self._QtGui.real_QApplication.exec_(self.app)
        exec_ = getattr(self._QtGui.real_QApplication, "exec", None)
        if exec_ is None:
            exec_ = self._QtGui.real_QApplication.exec_
        try:
            try_again = False
            exec_(self.app)
        except TypeError:
            try_again = True
        if try_again:
            exec_()
    def quit(self):
        # A nicer way to quit
        self._QtGui.real_QApplication.quit()
class App_pyqt6(App_qt):
    """Hijack the PyQt6 mainloop."""
    def importCoreAndGui(self):
        # Try importing qt
        import PyQt6  # noqa
        from PyQt6 import QtGui, QtCore, QtWidgets  # noqa
        return QtWidgets, QtCore  # QApp sits on QtWidgets
class App_pyqt5(App_qt):
    """Hijack the PyQt5 mainloop."""
    def importCoreAndGui(self):
        # Try importing qt
        import PyQt5  # noqa
        from PyQt5 import QtGui, QtCore, QtWidgets  # noqa
        return QtWidgets, QtCore  # QApp sits on QtWidgets
class App_pyqt4(App_qt):
    """Hijack the PyQt4 mainloop."""
    def importCoreAndGui(self):
        # Try importing qt
        import PyQt4  # noqa
        from PyQt4 import QtGui, QtCore
        return QtGui, QtCore
class App_pyside6(App_qt):
    """Hijack the PySide6 mainloop."""
    def importCoreAndGui(self):
        # Try importing qt
        import PySide6  # noqa
        from PySide6 import QtGui, QtCore, QtWidgets  # noqa
        return QtWidgets, QtCore  # QApp sits on QtWidgets
class App_pyside2(App_qt):
    """Hijack the PySide2 mainloop."""
    def importCoreAndGui(self):
        # Try importing qt
        import PySide2  # noqa
        from PySide2 import QtGui, QtCore, QtWidgets  # noqa
        return QtWidgets, QtCore  # QApp sits on QtWidgets
class App_pyside(App_qt):
    """Hijack the PySide mainloop."""
    def importCoreAndGui(self):
        # Try importing qt
        import PySide  # noqa
        from PySide import QtGui, QtCore
        return QtGui, QtCore
class App_wx(App_base):
    """Hijack the wxWidgets mainloop."""
    def __init__(self):
        # Try importing
        try:
            import wx
        except ImportError:
            # For very old versions of WX
            import wxPython as wx
        # Create dummy mainloop to replace original mainloop
        def dummy_mainloop(*args, **kw):
            print_mainloop_warning()
        # Depending on version, replace mainloop
        ver = wx.__version__
        orig_mainloop = None
        if ver[:3] >= "2.5":
            if hasattr(wx, "_core_"):
                core = getattr(wx, "_core_")
            elif hasattr(wx, "_core"):
                core = getattr(wx, "_core")
            else:
                raise ImportError
            orig_mainloop = core.PyApp_MainLoop
            core.PyApp_MainLoop = dummy_mainloop
        elif ver[:3] == "2.4":
            orig_mainloop = wx.wxc.wxPyApp_MainLoop
            wx.wxc.wxPyApp_MainLoop = dummy_mainloop
        else:
            # Unable to find either wxPython version 2.4 or >= 2.5."
            raise ImportError
        self._orig_mainloop = orig_mainloop
        # Store package wx
        self.wx = wx
        # Get and store the app instance to process events
        app = wx.GetApp()
        if app is None:
            app = wx.App(False)
        self.app = app
        # Notify that we integrated the event loop
        self.app._in_event_loop = "Pyzo"
        wx._in_event_loop = "Pyzo"
    def process_events(self):
        wx = self.wx
        # This bit is really needed
        old = wx.EventLoop.GetActive()
        eventLoop = wx.EventLoop()
        wx.EventLoop.SetActive(eventLoop)
        while eventLoop.Pending():
            eventLoop.Dispatch()
        # Process and reset
        self.app.ProcessIdle()  # otherwise frames do not close
        wx.EventLoop.SetActive(old)
```

```
class App_gtk(App_base):
    """Modifies pyGTK's mainloop with a dummy so user code does not
    block IPython.  processing events is done using the module'
    main_iteration function.
    """
    def __init__(self):
        # Try importing gtk
        import gtk
        # Replace mainloop with a dummy
        def dummy_mainloop(*args, **kwargs):
            print_mainloop_warning()
        gtk.mainloop = dummy_mainloop
        gtk.main = dummy_mainloop
        # Replace main_quit with a dummy too
        def dummy_quit(*args, **kwargs):
            pass
        gtk.main_quit = dummy_quit
        gtk.mainquit = dummy_quit
        # Make sure main_iteration exists even on older versions
        if not hasattr(gtk, "main_iteration"):
            gtk.main_iteration = gtk.mainiteration
        # Store 'app object'
        self.app = gtk
        # Notify that we integrated the event loop
        self.app._in_event_loop = "Pyzo"
    def process_events(self):
        gtk = self.app
        while gtk.events_pending():
            gtk.main_iteration(False)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module pyzokernel.interpreter
Implements the Pyzo interpreter.
Notes on IPython
----------------
We integrate IPython via the IPython.core.interactiveshell.InteractiveShell.
  * The namespace is set to __main__
  * We call its run_cell method to execute code
  * Debugging/breakpoints are "enabled using the pre_run_code_hook
  * Debugging occurs in our own debugger
  * GUI integration is all handled by pyzo
  * We need special prompts for IPython input
"""
import os
import sys
import time
import logging
import platform
import struct
import shlex
from codeop import CommandCompiler
import traceback
import keyword
import inspect  # noqa - Must be in this namespace
import bdb
from distutils.version import LooseVersion as LV
import linecache
import yoton
from pyzokernel import guiintegration, printDirect
from pyzokernel.magic import Magician
from pyzokernel.debug import Debugger
# Init last traceback information
sys.last_type = None
sys.last_value = None
sys.last_traceback = None
# Set Python version and get some names
PYTHON_VERSION = sys.version_info[0]
if PYTHON_VERSION < 3:
    ustr = unicode  # noqa
    bstr = str
    input = raw_input  # noqa
else:
    ustr = str
    bstr = bytes
class PS1:
    """Dynamic prompt for PS1. Show IPython prompt if available, and
    show current stack frame when debugging.
    """
    def __init__(self, pyzo):
        self._pyzo = pyzo
    def __str__(self):
        if self._pyzo._dbFrames:
            # When debugging, show where we are, do not use IPython prompt
            preamble = "(" + self._pyzo._dbFrameName + ")"
            return "\n\x1b[0;32m%s>>>\x1b[0m " % preamble
        elif self._pyzo._ipython:
            # IPython prompt
            return "\n\x1b[0;32mIn [\x1b[1;32m%i\x1b[0;32m]:\x1b[0m " % (
                self._pyzo._ipython.execution_count
            )
            # return 'In [%i]: ' % (self._ipython.execution_count)
        else:
            # Normal Python prompt
            return "\n\x1b[0;32m>>>\x1b[0m "
class PS2:
    """Dynamic prompt for PS2."""
    def __init__(self, pyzo):
        self._pyzo = pyzo
    def __str__(self):
        if self._pyzo._dbFrames:
            # When debugging, show where we are, do not use IPython prompt
            preamble = "(" + self._pyzo._dbFrameName + ")"
            return "\x1b[0;32m%s...\x1b[0m " % preamble
        elif self._pyzo._ipython:
            # Dots ala IPython
            nspaces = len(str(self._pyzo._ipython.execution_count)) + 2
            return "\x1b[0;32m%s...:\x1b[0m " % (nspaces * " ")
        else:
            # Just dots
            return "\x1b[0;32m...\x1b[0m "
class PyzoInterpreter:
    """PyzoInterpreter
    The pyzo interpreter is the part that makes the pyzo kernel interactive.
    It executes code, integrates the GUI toolkit, parses magic commands, etc.
    The pyzo interpreter has been designed to emulate the standard interactive
    Python console as much as possible, but with a lot of extra goodies.
    There is one instance of this class, stored at sys._pyzoInterpreter and
    at the __pyzo__ variable in the global namespace.
    The global instance has a couple of interesting attributes:
      * context: the yoton Context instance at the kernel (has all channels)
      * introspector: the introspector instance (a subclassed yoton.RepChannel)
      * magician: the object that handles the magic commands
      * guiApp: a wrapper for the integrated GUI application
    """
    # Simular working as code.InteractiveConsole. Some code was copied, but
    # the following things are changed:
    # - prompts are printed in the err stream, like the default interpreter does
    # - uses an asynchronous read using the yoton interface
    # - support for hijacking GUI toolkits
    # - can run large pieces of code
    # - support post mortem debugging
    # - support for magic commands
    def __init__(self, locals, filename="<console>"):
        # Init variables for locals and globals (globals only for debugging)
        self.locals = locals
        self.globals = None
        # Store filename
        self._filename = filename
        # Store ref of locals that is our main
        self._main_locals = locals
        # Flag to ignore sys exit, to allow running some scripts
        # interactively, even if they call sys.exit.
        self.ignore_sys_exit = False
        # Information for debugging. If self._dbFrames, we're in debug mode
        # _dbFrameIndex starts from 1
        self._dbFrames = []
        self._dbFrameIndex = 0
        self._dbFrameName = ""
        # Init datase to store source code that we execute
        self._codeCollection = ExecutedSourceCollection()
        # Init buffer to deal with multi-line command in the shell
        self._buffer = []
        # Create compiler
        if sys.platform.startswith("java"):
            import compiler
            self._compile = compiler.compile  # or 'exec' does not work
        else:
            self._compile = CommandCompiler()
        # Instantiate magician and tracer
        self.magician = Magician()
        self.debugger = Debugger()
        # To keep track of whether to send a new prompt, and whether more
        # code is expected.
        self.more = 0
        self.newPrompt = True
        # Code and script to run on first iteration
        self._codeToRunOnStartup = None
        self._scriptToRunOnStartup = None
        # Remove "THIS" directory from the PYTHONPATH
        # to prevent unwanted imports. Same for pyzokernel dir
        thisPath = os.getcwd()
        for p in [thisPath, os.path.join(thisPath, "pyzokernel")]:
            while p in sys.path:
                sys.path.remove(p)
    def run(self):
        """Run (start the mainloop)
        Here we enter the main loop, which is provided by the guiApp.
        This event loop calls process_commands on a regular basis.
        We may also enter the debug intereaction loop, either from a
        request for post-mortem debugging, or *during* execution by
        means of a breakpoint. When in this debug-loop, the guiApp event
        loop lays still, but the debug-loop does call process-commands
        for user interaction.
        When the user wants to quit, SystemExit is raised (one way or
        another). This is detected in process_commands and the exception
        instance is stored in self._exitException. Then the debug-loop
        is stopped if necessary, and the guiApp is told to stop its event
        loop.
        And that brings us back here, where we exit using in order of
        preference: self._exitException, the exception with which the
        event loop was exited (if any), or a new exception.
        """
        # Prepare
        self._prepare()
        self._exitException = None
        # Enter main
        try:
            self.guiApp.run(self.process_commands)
        except SystemExit:
            # Set self._exitException if it is not set yet
            type, value, tb = sys.exc_info()
            del tb
            if self._exitException is None:
                self._exitException = value
        # Exit
        if self._exitException is None:
            self._exitException = SystemExit()
        raise self._exitException
    def _prepare(self):
        """Prepare for running the main loop.
        Here we do some initialization like obtaining the startup info,
        creating the GUI application wrapper, etc.
        """
        # Reset debug status
        self.debugger.writestatus()
        # Get startup info (get a copy, or setting the new version wont
trigger!)
        while self.context._stat_startup.recv() is None:
            time.sleep(0.02)
        self.startup_info = startup_info =
self.context._stat_startup.recv().copy()
        # Set startup info (with additional info)
        if sys.platform.startswith("java"):
            import __builtin__ as builtins  # Jython
        else:
            builtins = __builtins__
        if not isinstance(builtins, dict):
            builtins = builtins.__dict__
        startup_info["builtins"] = [builtin for builtin in builtins.keys()]
        startup_info["version"] = tuple(sys.version_info)
        startup_info["keywords"] = keyword.kwlist
        # Update startup info, we update again at the end of this method
        self.context._stat_startup.send(startup_info.copy())
        # Prepare the Python environment
        self._prepare_environment(startup_info)
        # Run startup code (before loading GUI toolkit or IPython
        self._run_startup_code(startup_info)
        # Write Python banner (to stdout)
        thename = "Python"
        if sys.version_info[0] == 2:
            thename = "Legacy Python"
        if "__pypy__" in sys.builtin_module_names:
            thename = "Pypy"
        if sys.platform.startswith("java"):
            thename = "Jython"
            # Jython cannot do struct.calcsize("P")
            import java.lang
            real_plat = java.lang.System.getProperty("os.name").lower()
            plat = "%s/%s" % (sys.platform, real_plat)
        elif sys.platform.startswith("win"):
            NBITS = 8 * struct.calcsize("P")
            plat = "Windows (%i bits)" % NBITS
        else:
            NBITS = 8 * struct.calcsize("P")
            plat = "%s (%i bits)" % (sys.platform, NBITS)
        printDirect(
            "%s %s on %s.\n" % (thename, sys.version.split("[")[0].rstrip(),
plat)
        )
        # Integrate GUI
        guiName, guiError = self._integrate_gui(startup_info)
        # Write pyzo part of banner (including what GUI loop is integrated)
        if True:
            pyzoBanner = "This is the Pyzo interpreter"
        if guiError:
            pyzoBanner += ". " + guiError + "\n"
        elif guiName:
            pyzoBanner += " with integrated event loop for "
            pyzoBanner += guiName + ".\n"
        else:
            pyzoBanner += ".\n"
        printDirect(pyzoBanner)
        # Try loading IPython
        if startup_info.get("ipython", "").lower() in ("", "no", "false"):
            self._ipython = None
        else:
            try:
                self._load_ipyhon()
            except Exception:
                type, value, tb = sys.exc_info()
                del tb
                printDirect("IPython could not be loaded: %s\n" % str(value))
                self._ipython = None
                startup_info["ipython"] = "no"
            if not self._ipython:
                startup_info["ipython"] = "no"
        # Set prompts
        sys.ps1 = PS1(self)
        sys.ps2 = PS2(self)
        # Notify about project path
        projectPath = startup_info["projectPath"]
        if projectPath:
            printDirect("Prepending the project path %r to sys.path\n" %
projectPath)
        # Write tips message.
        if self._ipython:
            import IPython
            printDirect(
                "\nUsing IPython %s -- An enhanced Interactive Python.\n"
                % IPython.__version__
            )
            printDirect(
                "?         -> Introduction and overview of IPython's
features.\n"
                "%quickref -> Quick reference.\n"
                "help      -> Python's own help system.\n"
                "object?   -> Details about 'object', "
                "use 'object??' for extra details.\n"
            )
        else:
            printDirect(
                "Type 'help' for help, " + "type '?' for a list of *magic*
commands.\n"
            )
        # Notify the running of the script
        if self._scriptToRunOnStartup:
            printDirect(
                '\x1b[0;33mRunning script: "'
                + self._scriptToRunOnStartup
                + '"\x1b[0m\n'
            )
        # Prevent app nap on OSX 9.2 and up
        # The _nope module is taken from MINRK's appnope package
        if sys.platform == "darwin" and LV(platform.mac_ver()[0]) >= LV("10.9"):
            from pyzokernel import _nope
            _nope.nope()
        # Setup post-mortem debugging via appropriately logged exceptions
        class PMHandler(logging.Handler):
            def emit(self, record):
                if record.exc_info:
                    sys.last_type, sys.last_value, sys.last_traceback =
record.exc_info
                return record
        # Setup logging
        root_logger = logging.getLogger()
        if not root_logger.handlers:
            root_logger.addHandler(logging.StreamHandler())
        root_logger.addHandler(PMHandler())
        # Warn when logging.basicConfig is used (see issue #645)
        def basicConfigDoesNothing(*args, **kwargs):
            logging.warn(
                "Pyzo already added handlers to the root handler, "
                + "so logging.basicConfig() does nothing."
            )
        try:
            logging.basicConfig = basicConfigDoesNothing
        except Exception:
            pass
        # Update startup info
        self.context._stat_startup.send(startup_info)
    def _prepare_environment(self, startup_info):
        """Prepare the Python environment. There are two possibilities:
        either we run a script or we run interactively.
        """
        # Get whether we should (and can) run as script
        scriptFilename = startup_info["scriptFile"]
        if scriptFilename:
            if not os.path.isfile(scriptFilename):
                printDirect('Invalid script file: "' + scriptFilename + '"\n')
                scriptFilename = None
        # Get project path
        projectPath = startup_info["projectPath"]
        if scriptFilename.endswith(".ipynb"):
            # Run Jupyter notebook
            import notebook.notebookapp
            sys.argv = ["jupyter_notebook", scriptFilename]
            sys.exit(notebook.notebookapp.main())
        elif scriptFilename:
            # RUN AS SCRIPT
            # Set __file__  (note that __name__ is already '__main__')
            self.locals["__file__"] = scriptFilename
            # Set command line arguments
            sys.argv[:] = []
            sys.argv.append(scriptFilename)
            sys.argv.extend(shlex.split(startup_info.get("argv", "")))
            # Insert script directory to path
            theDir = os.path.abspath(os.path.dirname(scriptFilename))
            if theDir not in sys.path:
                sys.path.insert(0, theDir)
            if projectPath is not None:
                sys.path.insert(0, projectPath)
            # Go to script dir
            os.chdir(os.path.dirname(scriptFilename))
            # Run script later
            self._scriptToRunOnStartup = scriptFilename
        else:
            # RUN INTERACTIVELY
            # No __file__ (note that __name__ is already '__main__')
            self.locals.pop("__file__", "")
            # Remove all command line arguments, set first to empty string
            sys.argv[:] = []
            sys.argv.append("")
            sys.argv.extend(shlex.split(startup_info.get("argv", "")))
            # Insert current directory to path
            sys.path.insert(0, "")
            if projectPath:
                sys.path.insert(0, projectPath)
            # Go to start dir
            startDir = startup_info["startDir"]
            if startDir and os.path.isdir(startDir):
                os.chdir(startDir)
            else:
                os.chdir(os.path.expanduser("~"))  # home dir
    def _run_startup_code(self, startup_info):
        """Execute the startup code or script."""
        # Run startup script (if set)
        script = startup_info["startupScript"]
        # Should we use the default startupScript?
        if script == "$PYTHONSTARTUP":
            script = os.environ.get("PYTHONSTARTUP", "")
        if "\n" in script:
            # Run code later or now
            linesBefore = []
            linesAfter = script.splitlines()
            while linesAfter:
                if linesAfter[0].startswith("# AFTER_GUI"):
                    linesAfter.pop(0)
                    break
                linesBefore.append(linesAfter.pop(0))
            scriptBefore = "\n".join(linesBefore)
            self._codeToRunOnStartup = "\n".join(linesAfter)
            if scriptBefore.strip():  # don't trigger when only empty lines
                self.context._stat_interpreter.send("Busy")
                msg = {"source": scriptBefore, "fname": "<startup>", "lineno":
0}
                self.runlargecode(msg, True)
        elif script and os.path.isfile(script):
            # Run script
            self.context._stat_interpreter.send("Busy")
            self.runfile(script)
        else:
            # Nothing to run
            pass
    def _integrate_gui(self, startup_info):
        """Integrate event loop of GUI toolkit (or use pure Python
        event loop).
        """
        self.guiApp = guiintegration.App_nogui()
        self.guiName = guiName = startup_info["gui"].upper()
        guiError = ""
        try:
            if guiName in ["", "NONE"]:
                guiName = ""
            elif guiName == "AUTO":
                for tryName, tryApp in [
                    ("PYSIDE6", guiintegration.App_pyside6),
                    ("PYQT6", guiintegration.App_pyqt6),
                    ("PYSIDE2", guiintegration.App_pyside2),
                    ("PYQT5", guiintegration.App_pyqt5),
                    ("PYSIDE", guiintegration.App_pyside),
                    ("PYQT4", guiintegration.App_pyqt4),
                    # ('WX', guiintegration.App_wx),
                    ("ASYNCIO", guiintegration.App_asyncio_new),
                    ("TK", guiintegration.App_tk),
                ]:
                    try:
```

```
                    self.guiApp = tryApp()
                except Exception:
                    continue
                guiName = tryName
                break
            else:
                guiName = ""
        elif guiName == "ASYNCIO":
            self.guiApp = guiintegration.App_asyncio_new()
        elif guiName == "TK":
            self.guiApp = guiintegration.App_tk()
        elif guiName == "WX":
            self.guiApp = guiintegration.App_wx()
        elif guiName == "TORNADO":
            self.guiApp = guiintegration.App_tornado()
        elif guiName == "PYSIDE6":
            self.guiApp = guiintegration.App_pyside6()
        elif guiName == "PYSIDE2":
            self.guiApp = guiintegration.App_pyside2()
        elif guiName == "PYSIDE":
            self.guiApp = guiintegration.App_pyside()
        elif guiName in ["PYQT6", "QT6"]:
            self.guiApp = guiintegration.App_pyqt6()
        elif guiName in ["PYQT5", "QT5"]:
            self.guiApp = guiintegration.App_pyqt5()
        elif guiName in ["PYQT4", "QT4"]:
            self.guiApp = guiintegration.App_pyqt4()
        elif guiName == "FLTK":
            self.guiApp = guiintegration.App_fltk()
        elif guiName == "GTK":
            self.guiApp = guiintegration.App_gtk()
        else:
            guiError = "Unkown gui: %s" % guiName
            guiName = ""
    except Exception:  # Catch any error
        # Get exception info (we do it using sys.exc_info() because
        # we cannot catch the exception in a version independent way.
        type, value, tb = sys.exc_info()
        del tb
        guiError = "Failed to integrate event loop for %s: %s" % (
            guiName,
            str(value),
        )
    return guiName, guiError
def _load_ipyhon(self):
    """Try loading IPython shell. The result is set in self._ipython
    (can be None if IPython not available).
    """
    # Init
    self._ipython = None
    import __main__
    # Try importing IPython
    try:
        import IPython
    except ImportError:
        return
    # Version ok?
    if IPython.version_info < (1,):
        return
    # Create an IPython shell
    from IPython.core.interactiveshell import InteractiveShell
    self._ipython = InteractiveShell(user_module=__main__)
    # Set some hooks / event callbacks
    # Run hook (pre_run_code_hook is depreacted in 2.0)
    pre_run_cell_hook = self.ipython_pre_run_cell_hook
    if IPython.version_info < (2,):
        self._ipython.set_hook("pre_run_code_hook", pre_run_cell_hook)
    else:
        self._ipython.events.register("pre_run_cell", pre_run_cell_hook)
    # Other hooks
    self._ipython.set_hook("editor", self.ipython_editor_hook)
    self._ipython.set_custom_exc((bdb.BdbQuit,), self.dbstop_handler)
    # Some patching
    self._ipython.ask_exit = self.ipython_ask_exit
    # Make output be shown on Windows
    if sys.platform.startswith("win"):
        # IPython wraps std streams just like we do below, but
        # pyreadline adds *another* wrapper, which is where it
        # goes wrong. Here we set it back to bypass pyreadline.
        if IPython.version_info < (
            8,
        ):  # corrects a problem with IOStream from IPython 8.x.x
            from IPython.utils import io
            io.stdin = io.IOStream(sys.stdin)
            io.stdout = io.IOStream(sys.stdout)
            io.stderr = io.IOStream(sys.stderr)
        # Ipython uses msvcrt e.g. for pausing between pages
        # but this does not work in pyzo
        import msvcrt
        msvcrt.getwch = msvcrt.getch = input  # input is deffed above
def process_commands(self):
    """Do one iteration of processing commands (the REPL)."""
    try:
        self._process_commands()
    except SystemExit:
        # It may be that we should ignore sys exit now...
        if self.ignore_sys_exit:
            self.ignore_sys_exit = False  # Never allow more than once
            return
        # Get and store the exception so we can raise it later
        type, value, tb = sys.exc_info()
        del tb
        self._exitException = value
        # Stop debugger if it is running
        self.debugger.stopinteraction()
        # Exit from interpreter. Exit in the appropriate way
        self.guiApp.quit()  # Is sys.exit() by default
def _process_commands(self):
    # Run startup code/script inside the loop (only the first time)
    # so that keyboard interrupt will work
    if self._codeToRunOnStartup:
        self.context._stat_interpreter.send("Busy")
        self._codeToRunOnStartup, tmp = None, self._codeToRunOnStartup
        self._runlines(tmp, filename="<startup_after_gui>", symbol="exec")
    if self._scriptToRunOnStartup:
        self.context._stat_interpreter.send("Busy")
        self._scriptToRunOnStartup, tmp = None, self._scriptToRunOnStartup
        self.runfile(tmp)
    # Flush real stdout / stderr
    sys.__stdout__.flush()
    sys.__stderr__.flush()
    # Set status and prompt?
    # Prompt is allowed to be an object with __str__ method
    if self.newPrompt:
        self.newPrompt = False
        ps = [sys.ps1, sys.ps2][bool(self.more)]
        self.context._strm_prompt.send(str(ps))
    if True:
        # Determine state. The message is really only send
        # when the state is different. Note that the kernelbroker
        # can also set the state ("Very busy", "Busy", "Dead")
        if self._dbFrames:
            self.context._stat_interpreter.send("Debug")
        elif self.more:
            self.context._stat_interpreter.send("More")
        else:
            self.context._stat_interpreter.send("Ready")
        self.context._stat_cd.send(os.getcwd())
    # Are we still connected?
    if sys.stdin.closed or not self.context.connection_count:
        # Exit from main loop.
        # This will raise SystemExit and will shut us down in the
        # most appropriate way
        sys.exit()
    # Get channel to take a message from
    ch = yoton.select_sub_channel(
        self.context._ctrl_command, self.context._ctrl_code
    )
    if ch is None:
        pass  # No messages waiting
    elif ch is self.context._ctrl_command:
        # Read command
        line1 = self.context._ctrl_command.recv(False)  # Command
        if line1:
            # Notify what we're doing
            self.context._strm_echo.send(line1)
            self.context._stat_interpreter.send("Busy")
            self.newPrompt = True
            # Convert command
            # (only a few magics are supported if IPython is active)
            line2 = self.magician.convert_command(line1.rstrip("\n"))
            # Execute actual code
            if line2 is not None:
                for line3 in line2.split("\n"):  # not splitlines!
                    self.more = self.pushline(line3)
            else:
                self.more = False
                self._resetbuffer()
    elif ch is self.context._ctrl_code:
        # Read larger block of code (dict)
        msg = self.context._ctrl_code.recv(False)
        if msg:
            # Notify what we're doing
            # (runlargecode() sends on stdin-echo)
            self.context._stat_interpreter.send("Busy")
            self.newPrompt = True
            # Execute code
            self.runlargecode(msg)
            # Reset more stuff
            self._resetbuffer()
            self.more = False
    else:
        # This should not happen, but if it does, just flush!
        ch.recv(False)
## Running code in various ways
# In all cases there is a call for compilecode and a call to execcode
def _resetbuffer(self):
    """Reset the input buffer."""
    self._buffer = []
def pushline(self, line):
    """Push a line to the interpreter.
    The line should not have a trailing newline; it may have
    internal newlines.  The line is appended to a buffer and the
    interpreter's _runlines() method is called with the
    concatenated contents of the buffer as source.  If this
    indicates that the command was executed or invalid, the buffer
    is reset; otherwise, the command is incomplete, and the buffer
    is left as it was after the line was appended.  The return
    value is 1 if more input is required, 0 if the line was dealt
    with in some way (this is the same as _runlines()).
    """
    # Get buffer, join to get source
    buffer = self._buffer
    buffer.append(line)
    source = "\n".join(buffer)
    # Clear buffer and run source
    self._resetbuffer()
    more = self._runlines(source, self._filename)
    # Create buffer if needed
    if more:
        self._buffer = buffer
    return more
def _runlines(self, source, filename="<input>", symbol="single"):
    """Compile and run some source in the interpreter.
    Arguments are as for compile_command().
    One several things can happen:
    1) The input is incorrect; compile_command() raised an
    exception (SyntaxError or OverflowError).  A syntax traceback
    will be printed by calling the showsyntaxerror() method.
    2) The input is incomplete, and more input is required;
    compile_command() returned None.  Nothing happens.
    3) The input is complete; compile_command() returned a code
    object.  The code is executed by calling self.execcode() (which
    also handles run-time exceptions, except for SystemExit).
    The return value is True in case 2, False in the other cases (unless
    an exception is raised).  The return value can be used to
    decide whether to use sys.ps1 or sys.ps2 to prompt the next
    line.
    """
    use_ipython = self._ipython and not self._dbFrames
    # Try compiling.
    # The IPython kernel does not handle incomple lines, so we check
    # that ourselves ...
    error = None
    try:
        code = self.compilecode(source, filename, symbol)
    except (OverflowError, SyntaxError, ValueError):
        error = sys.exc_info()[1]
        code = False
    if use_ipython:
        if code is None:
            # Case 2
            # self._ipython.run_cell('', True)
            return True
        else:
            # Case 1 and 3 handled by IPython
            self._ipython.run_cell(source, True, False)
            return False
    else:
        if code is None:
            # Case 2
            return True
        elif not code:
            # Case 1, a bit awkward way to show the error, but we need
            # to call showsyntaxerror in an exception handler.
            try:
                raise error
            except Exception:
```

```
                self.showsyntaxerror(filename)
            return False
        else:
            # Case 3
            self.execcode(code)
            return False
    def runlargecode(self, msg, silent=False):
        """To execute larger pieces of code."""
        # Get information
        source, fname, lineno = msg["source"], msg["fname"], msg["lineno"]
        cellName = msg.get("cellName", "")
        source += "\n"
        # Change directory?
        if msg.get("changeDir", False) and os.path.isfile(fname):
            d = os.path.normpath(os.path.normcase(os.path.dirname(fname)))
            if d != os.getcwd():
                os.chdir(d)
        # Construct notification message
        lineno1 = lineno + 1
        lineno2 = lineno + source.count("\n")
        fname_show = fname
        if not fname.startswith("<"):
            fname_show = os.path.split(fname)[1]
        if cellName == fname:
            runtext = '(executing file "%s")\n' % fname_show
        elif cellName:
            runtext = '(executing cell "%s" (line %i of "%s"))\n' % (
                cellName.strip(),
                lineno1,
                fname_show,
            )
            # Try to get the last expression printed in the cell.
            try:
                import ast
                tree = ast.parse(source, fname, "exec")
                if (
                    isinstance(tree.body[-1], ast.Expr)
                    and tree.body[-1].col_offset == 0
                ):
                    e = tree.body[-1]
                    lines = source.splitlines()
                    lines[e.lineno - 1] = "_=\\\n" + lines[e.lineno - 1]
                    source2 = (
                        "\n".join(lines).rstrip()
                        + "\nif _ is not None:\n  print(repr(_))\n"
                    )
                    ast.parse(
                        source2, fname, "exec"
                    )  # This is to make sure it still compiles
                    source = source2
            except Exception:
                pass
        elif lineno1 == lineno2:
            runtext = '(executing line %i of "%s")\n' % (lineno1, fname_show)
        else:
            runtext = '(executing lines %i to %i of "%s")\n' % (
                lineno1,
                lineno2,
                fname_show,
            )
        # Notify IDE
        if not silent:
            self.context._strm_echo.send("\x1b[0;33m%s\x1b[0m" % runtext)
            # Increase counter
            if self._ipython:
                self._ipython.execution_count += 1
        # Bring fname to the canonical form so that pdb recognizes the 
breakpoints,
        # otherwise filename r"C:\..." would be different from canonical form 
r"c:\..."
        fname = self.debugger.canonic(fname)
        # Put the line number in the filename (if necessary)
        # Note that we could store the line offset in the _codeCollection,
        # but then we cannot retrieve it for syntax errors.
        if lineno:
            fname = "%s+%i" % (fname, lineno)
        # Try compiling the source
        code = None
        try:
            # Compile
            code = self.compilecode(source, fname, "exec")
        except (OverflowError, SyntaxError, ValueError):
            self.showsyntaxerror(fname)
            return
        if code:
            # Store the source using the (id of the) code object as a key
            self._codeCollection.store_source(code, source)
            # Execute the code
            self.execcode(code)
        else:
            # Incomplete code
            self.write("Could not run code because it is incomplete.\n")
    def runfile(self, fname):
        """To execute the startup script."""
        # Get text (make sure it ends with a newline)
        try:
            bb = open(fname, "rb").read()
            encoding = "UTF-8"
            firstline = bb.split("\n".encode(), 1)[0].decode("ascii", "ignore")
            if firstline.startswith("#") and "coding" in firstline:
                encoding = firstline.split("coding", 1)[-1].strip(" \t\r\n:=-*")
            source = bb.decode(encoding)
        except Exception:
            printDirect(
                "Could not read script (decoding using %s): %r\n" % (encoding,
fname)
            )
            return
        try:
            source = source.replace("\r\n", "\n").replace("\r", "\n")
            if source[-1] != "\n":
                source += "\n"
        except Exception:
            printDirect('Could not execute script: "' + fname + '"\n')
            return
        # Try compiling the source
        code = None
        try:
            # Compile
            code = self.compilecode(source, fname, "exec")
        except (OverflowError, SyntaxError, ValueError):
            time.sleep(0.2)  # Give stdout time to be send
            self.showsyntaxerror(fname)
            return
        if code:
            # Store the source using the (id of the) code object as a key
            self._codeCollection.store_source(code, source)
            # Execute the code
            self.execcode(code)
        else:
            # Incomplete code
            self.write("Could not run code because it is incomplete.\n")
    def compilecode(self, source, filename, mode, *args, **kwargs):
        """Compile source code.
        Will mangle coding definitions on first two lines.
        * This method should be called with Unicode sources.
        * Source newlines should consist only of LF characters.
        """
        # This method solves pyzo issue 22
        # Split in first two lines and the rest
        parts = source.split("\n", 2)
        # Replace any coding definitions
        ci = "coding is"
        contained_coding = False
        for i in range(len(parts) - 1):
            tmp = parts[i]
            if tmp and tmp[0] == "#" and "coding" in tmp:
                contained_coding = True
                parts[i] = tmp.replace("coding=", ci).replace("coding:", ci)
        # Combine parts again (if necessary)
        if contained_coding:
            source = "\n".join(parts)
        # Convert filename to UTF-8 if Python version < 3
        if PYTHON_VERSION < 3:
            filename = filename.encode("utf-8")
        # Compile
        return self._compile(source, filename, mode, *args, **kwargs)
    def execcode(self, code):
        """Execute a code object.
        When an exception occurs, self.showtraceback() is called to
        display a traceback.  All exceptions are caught except
        SystemExit, which is reraised.
        A note about KeyboardInterrupt: this exception may occur
        elsewhere in this code, and may not always be caught.  The
        caller should be prepared to deal with it.
        The globals variable is used when in debug mode.
        """
        try:
            if self._dbFrames:
                self.apply_breakpoints()
                exec(code, self.globals, self.locals)
            else:
                # Turn debugger on at this point. If there are no breakpoints,
                # the tracing is disabled for better performance.
                self.apply_breakpoints()
                self.debugger.set_on()
                exec(code, self.locals)
        except bdb.BdbQuit:
            self.dbstop_handler()
        except Exception:
            time.sleep(0.2)  # Give stdout some time to send data
            self.showtraceback()
        except KeyboardInterrupt:  # is a BaseException, not an Exception
            time.sleep(0.2)
            self.showtraceback()
    def apply_breakpoints(self):
        """Breakpoints are updated at each time a command is given,
        including commands like "db continue".
        """
        try:
            breaks = self.context._stat_breakpoints.recv()
            if self.debugger.breaks:
                self.debugger.clear_all_breaks()
            if breaks:  # Can be None
                for fname in breaks:
                    for linenr in breaks[fname]:
                        self.debugger.set_break(fname, linenr)
        except Exception:
            type, value, tb = sys.exc_info()
            del tb
            print("Error while setting breakpoints: %s" % str(value))
    ## Handlers and hooks
    def ipython_pre_run_cell_hook(self, ipython=None):
        """Hook that IPython calls right before executing code."""
        self.apply_breakpoints()
        self.debugger.set_on()
    def ipython_editor_hook(self, ipython, filename, linenum=None, wait=True):
        # Correct line number for cell offset
        filename, linenum = self.correctfilenameandlineno(filename, linenum or
0)
        # Get action string
        if linenum:
            action = "open %i %s" % (linenum, os.path.abspath(filename))
        else:
            action = "open %s" % os.path.abspath(filename)
        # Send
        self.context._strm_action.send(action)
    def ipython_ask_exit(self):
        # Ask the user
        a = input("Do you really want to exit ([y]/n)? ")
        a = a or "y"
        # Close stdin if necessary
        if a.lower() == "y":
            sys.stdin._channel.close()
    def dbstop_handler(self, *args, **kwargs):
        print("Program execution stopped from debugger.")
    ## Writing and error handling
    def write(self, text):
        """Write errors."""
        sys.stderr.write(text)
    def showsyntaxerror(self, filename=None):
        """Display the syntax error that just occurred.
        This doesn't display a stack trace because there isn't one.
        If a filename is given, it is stuffed in the exception instead
        of what was there before (because Python's parser always uses
        "<string>" when reading from a string).
        Pyzo version: support to display the right line number,
        see doc of showtraceback for details.
        """
        # Get info (do not store)
        type, value, tb = sys.exc_info()
        del tb
        # Work hard to stuff the correct filename in the exception
        if filename and type is SyntaxError:
            try:
                # unpack information
                msg, (dummy_filename, lineno, offset, line) = value
                # correct line-number
                fname, lineno = self.correctfilenameandlineno(filename, lineno)
            except Exception:
                # Not the format we expect; leave it alone
                pass
            else:
                # Stuff in the right filename
                value = SyntaxError(msg, (fname, lineno, offset, line))
                sys.last_value = value
        # Show syntax error
        strList = traceback.format_exception_only(type, value)
        for s in strList:
            self.write(s)
    def showtraceback(self, useLastTraceback=False):
        """Display the exception that just occurred.
        We remove the first stack item because it is our own code.
        The output is written by self.write(), below.
        In the pyzo version, before executing a block of code,
        the filename is modified by appending " [x]". Where x is
        the index in a list that we keep, of tuples
        (sourcecode, filename, lineno).
        Here, showing the traceback, we check if we see such [x],
        and if so, we extract the line of code where it went wrong,
        and correct the lineno, so it will point at the right line
        in the editor if part of a file was executed. When the file
        was modified since the part in question was executed, the
        fileno might deviate, but the line of code shown shall
        always be correct...
        """
        # Traceback info:
        # tb_next -> go down the trace
        # tb_frame -> get the stack frame
        # tb_lineno -> where it went wrong
        #
        # Frame info:
        # f_back -> go up (towards caller)
        # f_code -> code object
        # f_locals -> we can execute code here when PM debugging
        # f_globals
        # f_trace -> (can be None) function for debugging? (
        #
        # The traceback module is used to obtain prints from the
        # traceback.
        try:
            if useLastTraceback:
                # Get traceback info from buffered
                type = sys.last_type
                value = sys.last_value
                tb = sys.last_traceback
            else:
                # Get exception information and remove first, since that's us
                type, value, tb = sys.exc_info()
                tb = tb.tb_next
                # Store for debugging, but only store if not in debug mode
                if not self._dbFrames:
                    sys.last_type = type
                    sys.last_value = value
                    sys.last_traceback = tb
            # Get traceback to correct all the line numbers
            # tblist = list  of (filename, line-number, function-name, text)
            tblist = traceback.extract_tb(tb)
            # Get frames
            frames = []
            while tb:
                frames.append(tb.tb_frame)
                tb = tb.tb_next
            # Walk through the list
            for i in range(len(tblist)):
                tbInfo = tblist[i]
                # Get filename and line number, init example
                fname, lineno = self.correctfilenameandlineno(tbInfo[0],
tbInfo[1])
                if not isinstance(fname, ustr):
                    fname = fname.decode("utf-8")
                example = tbInfo[3]
                # Reset info
                tblist[i] = (fname, lineno, tbInfo[2], example)
            # Format list
            strList = traceback.format_list(tblist)
            if strList:
                strList.insert(0, "Traceback (most recent call last):\n")
            strList.extend(traceback.format_exception_only(type, value))
            # Write traceback
            for s in strList:
                self.write(s)
            # Clean up (we cannot combine except and finally in Python <2.5
            tb = None
            frames = None
        except Exception:
            type, value, tb = sys.exc_info()
            tb = None
            frames = None
            t = "An error occured, but then another one when trying to write the
traceback: "
            t += str(value) + "\n"
            self.write(t)
    def correctfilenameandlineno(self, fname, lineno):
        """Given a filename and lineno, this function returns
        a modified (if necessary) version of the two.
        As example:
        "foo.py+7", 22  -> "foo.py", 29
        """
        j = fname.rfind("+")
        if j > 0:
            try:
                lineno += int(fname[j + 1 :])
                fname = fname[:j]
            except ValueError:
                pass
        return fname, lineno
class ExecutedSourceCollection:
    """Stores the source of executed pieces of code, so that the right
    traceback can be reproduced when an error occurs. The filename
    (including the +lineno suffix) is used as a key. We monkey-patch
    the linecache module so that we first try our cache to look up the
    lines. In that way we also allow third party modules (e.g. IPython)
    to get the lines for executed cells.
    """
    def __init__(self):
        self._cache = {}
        self._patch()
    def store_source(self, codeObject, source):
        self._cache[codeObject.co_filename] = source
    def _patch(self):
        def getlines(filename, module_globals=None):
            # !!! do not use "import" inside this function because it can
            # cause an infinite recursion loop with the import override in
            # module shiboken6 (PySide6) !!!
            # Try getting the source from our own cache,
            # otherwise fallback to linecache's own cache
            src = self._cache.get(filename, "")
            if src:
                return [line + "\n" for line in src.splitlines()]
            else:
                if module_globals is None:
                    return linecache._getlines(filename)  # only valid sig in
2.4
                else:
                    return linecache._getlines(filename, module_globals)
        # Monkey patch
        linecache._getlines = linecache.getlines
        linecache.getlines = getlines
        # I hoped this would remove the +lineno for IPython tracebacks,
        # but it doesn't
#         def extract_tb(tb, limit=None):
#             print('aasdasd')
#             import traceback
#             list1 = traceback._extract_tb(tb, limit)
#             list2 = []
#             for (filename, lineno, name, line) in list1:
#                 filename, lineno = 
sys._pyzoInterpreter.correctfilenameandlineno(filename, lineno)
#                 list2.append((filename, lineno, name, line))
#             return list2
#         
#         import traceback
#         traceback._extract_tb = traceback.extract_tb
#         traceback.extract_tb = extract_tb
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
import sys
import yoton
import inspect  # noqa - used in eval()
try:
    import thread  # Python 2
except ImportError:
    import _thread as thread  # Python 3
class PyzoIntrospector(yoton.RepChannel):
    """This is a RepChannel object that runs a thread to respond to
    requests from the IDE.
    """
    def _getNameSpace(self, name=""):
        """_getNameSpace(name='')
        Get the namespace to apply introspection in.
        If name is given, will find that name. For example sys.stdin.
        """
        # Get namespace
        NS1 = sys._pyzoInterpreter.locals
        NS2 = sys._pyzoInterpreter.globals
        if not NS2:
            NS = NS1
        else:
            NS = NS2.copy()
            NS.update(NS1)
        # Look up a name?
        if not name:
            return NS
        else:
            try:
                # Get object
                ob = eval(name, None, NS)
                # Get namespace for this object
                if isinstance(ob, dict):
                    NS = {}
                    for el in ob:
                        NS["[" + repr(el) + "]"] = ob[el]
                elif isinstance(ob, (list, tuple)):
                    NS = {}
                    count = -1
                    for el in ob:
                        count += 1
                        NS["[%i]" % count] = el
                else:
                    keys = dir(ob)
                    NS = {}
                    for key in keys:
                        try:
                            NS[key] = getattr(ob, key)
                        except Exception:
                            NS[key] = "<unknown>"
                # Done
                return NS
            except Exception:
                return {}
    def _getSignature(self, objectName):
        """_getSignature(objectName)
        Get the signature of builtin, function or method.
        Returns a tuple (signature_string, kind), where kind is a string
        of one of the above. When none of the above, both elements in
        the tuple are an empty string.
        """
        # if a class, get init
        # not if an instance! -> try __call__ instead
        # what about self?
        # Get valid object names
        parts = objectName.rsplit(".")
        objectNames = [".".join(parts[-i:]) for i in range(1, len(parts) + 1)]
        # find out what kind of function, or if a function at all!
        NS = self._getNameSpace()
        fun1 = eval("inspect.isbuiltin(%s)" % (objectName), None, NS)
        fun2 = eval("inspect.isfunction(%s)" % (objectName), None, NS)
        fun3 = eval("inspect.ismethod(%s)" % (objectName), None, NS)
        fun4 = False
        fun5 = False
        if not (fun1 or fun2 or fun3):
            # Maybe it's a class with an init?
            if eval("hasattr(%s,'__init__')" % (objectName), None, NS):
                objectName += ".__init__"
                fun4 = eval("inspect.ismethod(%s)" % (objectName), None, NS)
            #  Or a callable object?
            elif eval("hasattr(%s,'__call__')" % (objectName), None, NS):
                objectName += ".__call__"
                fun5 = eval("inspect.ismethod(%s)" % (objectName), None, NS)
        sigs = ""
        if True:
            # the first line in the docstring is usually the signature
            tmp = eval("%s.__doc__" % (objectNames[-1]), {}, NS)
            sigs = ""
            if tmp:
                sigs = tmp.splitlines()[0].strip()
            # Test if doc has signature
            hasSig = False
            for name in objectNames:  # list.append -> L.apend(objec) -- blabla
                name += "("
                if name in sigs:
                    hasSig = True
            # If not a valid signature, do not bother ...
            if (not hasSig) or (sigs.count("(") != sigs.count(")")):
                sigs = ""
        if fun1 or fun2 or fun3 or fun4 or fun5:
            if fun1:
                kind = "builtin"
            elif fun2:
                kind = "function"
            elif fun3:
                kind = "method"
            elif fun4:
                kind = "class"
            elif fun5:
                kind = "callable"
            if not sigs:
                # Use intospection
                funname = objectName.split(".")[-1]
                try:
                    tmp = eval(
                        "inspect.signature(%s)" % (objectName), None, NS
                    )  # py3.3
                    sigs = funname + str(tmp)
                except Exception:
                    try:
                        tmp = eval(
                            "inspect.getargspec(%s)" % (objectName), None, NS
                        )  # py2
                    except Exception:  # the above fails on 2.4 (+?) for
```

```
builtins
                        tmp = None
                        kind = ""
                    if tmp is not None:
                        args, varargs, varkw, defaults = tmp[:4]
                        # prepare defaults
                        if defaults is None:
                            defaults = ()
                        defaults = list(defaults)
                        defaults.reverse()
                        # make list (back to forth)
                        args2 = []
                        for i in range(len(args) - fun4):
                            arg = args.pop()
                            if i < len(defaults):
                                args2.insert(0, "%s=%s" % (arg, defaults[i]))
                            else:
                                args2.insert(0, arg)
                        # append varargs and kwargs
                        if varargs:
                            args2.append("*" + varargs)
                        if varkw:
                            args2.append("**" + varkw)
                        # append the lot to our  string
                        sigs = "%s(%s)" % (funname, ", ".join(args2))
        elif sigs:
            kind = "function"
        else:
            sigs = ""
            kind = ""
        return sigs, kind
    # todo: variant that also says whether it's a property/function/class/other
    def dir(self, objectName):
        """dir(objectName)
        Get list of attributes for the given name.
        """
        # sys.__stdout__.write('handling '+objectName+'\n')
        # sys.__stdout__.flush()
        # Get namespace
        NS = self._getNameSpace()
        # Init names
        names = set()
        # Obtain all attributes of the class
        try:
            command = "dir(%s.__class__)" % (objectName)
            d = eval(command, {}, NS)
        except Exception:
            pass
        else:
            names.update(d)
        # Obtain instance attributes
        try:
            command = "%s.__dict__.keys()" % (objectName)
            d = eval(command, {}, NS)
        except Exception:
            pass
        else:
            names.update(d)
        # That should be enough, but in case __dir__ is overloaded,
        # query that as well
        try:
            command = "dir(%s)" % (objectName)
            d = eval(command, {}, NS)
        except Exception:
            pass
        else:
            names.update(d)
        # Respond
        return list(names)
    def dir2(self, objectName):
        """dir2(objectName)
        Get variable names in currently active namespace plus extra information.
        Returns a list of tuple of strings: name, type, kind, repr.
        """
        try:
            name = ""
            names = []
            def storeInfo(name, val):
                # Determine type
                typeName = type(val).__name__
                # Determine kind
                kind = typeName
                if typeName != "type":
                    if (
                        hasattr(val, "__array__")
                        and hasattr(val, "dtype")
                        and hasattr(val, "shape")
                        and not hasattr(
                            val,
"if_this_is_an_attribute_then_there_are_likely_inf_attributes",
                        )
                    ):
                        kind = "array"
                    elif isinstance(val, list):
                        kind = "list"
                    elif isinstance(val, tuple):
                        kind = "tuple"
                # Determine representation
                if kind == "array":
                    tmp = "x".join([str(s) for s in val.shape])
                    if tmp:
                        values_repr = ""
                        if hasattr(val, "flat"):
                            for el in val.flat:
                                values_repr += ", " + repr(el)
                                if len(values_repr) > 70:
                                    values_repr = values_repr[:67] + ", …"
                                    break
                        repres = "<array %s %s: %s>" % (
                            tmp,
                            val.dtype.name,
                            values_repr,
                        )
                    elif val.size:
                        tmp = str(float(val))
                        if "int" in val.dtype.name:
                            tmp = str(int(val))
                        repres = "<array scalar %s (%s)>" % (val.dtype.name,
tmp)
                    else:
                        repres = "<array empty %s>" % (val.dtype.name)
                elif kind == "list":
                    values_repr = ""
                    for el in val:
                        values_repr += ", " + repr(el)
                        if len(values_repr) > 70:
                            values_repr = values_repr[:67] + ", …"
                            break
                    repres = "<%i-element list: %s>" % (len(val), values_repr)
                elif kind == "tuple":
                    values_repr = ""
                    for el in val:
                        values_repr += ", " + repr(el)
                        if len(values_repr) > 70:
                            values_repr = values_repr[:67] + ", …"
                            break
                    repres = "<%i-element tuple: %s>" % (len(val), values_repr)
                elif kind == "dict":
                    values_repr = ""
                    for k, v in val.items():
                        values_repr += ", " + repr(k) + ": " + repr(v)
                        if len(values_repr) > 70:
                            values_repr = values_repr[:67] + ", …"
                            break
                    repres = "<%i-item dict: %s>" % (len(val), values_repr)
                else:
                    repres = repr(val)
                    if len(repres) > 80:
                        repres = repres[:77] + "…"
                # Store
                tmp = (name, typeName, kind, repres)
                names.append(tmp)
            # Get locals
            NS = self._getNameSpace(objectName)
            for name in NS.keys():  # name can be a key in a dict, i.e. not str
                if hasattr(name, "startswith") and name.startswith("__"):
                    continue
                try:
                    storeInfo(str(name), NS[name])
                except Exception:
                    pass
            return names
        except Exception:
            return []
    def signature(self, objectName):
        """signature(objectName)
        Get signature.
        """
        try:
            text, kind = self._getSignature(objectName)
            return text
        except Exception:
            return None
    def doc(self, objectName):
        """doc(objectName)
        Get documentation for an object.
        """
        # Get namespace
        NS = self._getNameSpace()
        try:
            # collect docstring
            h_text = ""
            # Try using the class (for properties)
            try:
                className = eval("%s.__class__.__name__" % (objectName), {}, NS)
                if "." in objectName:
                    tmp = objectName.rsplit(".", 1)
                    tmp[1] += "."
                else:
                    tmp = [objectName, ""]
                if className not in [
                    "type",
                    "module",
                    "builtin_function_or_method",
                    "function",
                ]:
                    cmd = "%s.__class__.%s__doc__"
                    h_text = eval(cmd % (tmp[0], tmp[1]), {}, NS)
            except Exception:
                pass
            # Normal doc
            if not h_text:
                h_text = eval("%s.__doc__" % (objectName), {}, NS)
            # collect more data
            h_repr = eval("repr(%s)" % (objectName), {}, NS)
            try:
                h_class = eval("%s.__class__.__name__" % (objectName), {}, NS)
            except Exception:
                h_class = "unknown"
            # docstring can be None, but should be empty then
            if not h_text:
                h_text = ""
            # get and correct signature
            h_fun, kind = self._getSignature(objectName)
            if not h_fun:
                h_fun = ""  # signature not available
            # cut repr if too long
            if len(h_repr) > 200:
                h_repr = h_repr[:200] + "..."
            # replace newlines so we can separates the different parts
            h_repr = h_repr.replace("\n", "\r")
            # build final text
            text = "\n".join([objectName, h_class, h_fun, h_repr, h_text])
        except Exception:
            type, value, tb = sys.exc_info()
            del tb
            text = "\n".join(
                [objectName, "", "", "", "No help available. ", str(value)]
            )
        # Done
        return text
    def eval(self, command):
        """eval(command)
        Evaluate a command and return result.
        """
        # Get namespace
        NS = self._getNameSpace()
        try:
            # here globals is None, so we can look into sys, time, etc...
            return eval(command, None, NS)
        except Exception:
            return "Error evaluating: " + command
    def interrupt(self, command=None):
        """interrupt()
        Interrupt the main thread. This does not work if the main thread
        is running extension code.
        A bit of a hack to do this in the introspector, but it's the
        easeast way and prevents having to launch another thread just
        to wait for an interrupt/terminare command.
        Note that on POSIX we can send an OS INT signal, which is faster
        and maybe more effective in some situations.
        """
        thread.interrupt_main()
    def terminate(self, command=None):
        """terminate()
        Ask the kernel to terminate by closing the stdin.
        """
        sys.stdin._channel.close()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
"""
Magic commands for the Pyzo kernel.
No need to use printDirect here, magic commands are just like normal Python
commands, in the sense that they print something etc.
"""
import sys
import os
import re
import time
import inspect
import tokenize, token, keyword
import itertools
try:
    import io
except ImportError:  # Not on Python 2.4
    pass
# Set Python version and get some names
PYTHON_VERSION = sys.version_info[0]
if PYTHON_VERSION < 3:
    input = raw_input  # noqa
MESSAGE = """List of *magic* commands:
    ?               - show this message
    ?X or X?        - show docstring of X
    ??X or X??      - help(X)
    cd              - show current directory
    cd X            - change directory
    ls              - list current directory
    who             - list variables in current workspace
    whos            - list variables plus their class and representation
    timeit X        - times execution of command X
    open X          - open file X or the Python module that defines object X
    run X           - run file X
    install         - install a new package
    update          - update an installed package
    remove          - remove (i.e. uninstall) an installed package
    uninstall       - alias for remove
    conda           - manage packages using conda
    pip             - manage packages using pip
    db X            - debug commands
    cls             - clear screen
    notebook        - launch the Jupyter notebook
"""
TIMEIT_MESSAGE = """Time execution duration. Usage:
    timeit fun  # where fun is a callable
    timeit 'expression'
    timeit expression
    timeit 20 fun/expression  # tests 20 passes
"""
def _detect_equalbang(line):
    try:
        gtoks = tokenize.tokenize(io.BytesIO(line.encode("utf-8")).readline)
        # iteration on two successive elements,
https://docs.python.org/3/library/itertools.html#itertools-recipes
        first, second = itertools.tee(gtoks)
        next(second, None)
        for tok1, tok2 in zip(first, second):
            if (
                tok1.type == token.OP
                and tok1.string == "="
                and tok2.type == token.ERRORTOKEN
                and tok2.string == "!"
            ):
                return True
        return False
    except tokenize.TokenError:  # typically this means an unmatched parenthesis
        return False
def _should_not_interpret_as_magic(line):
    interpreter = sys._pyzoInterpreter
    # Check that line is not some valid python input
    try:
        # gets a list of 5-tuples, of which [0] is the type of token and [1] is
the token string
        ltok =
list(tokenize.tokenize(io.BytesIO(line.encode("utf-8")).readline))
    except tokenize.TokenError:
        # typically this means an unmatched parenthesis
        # (which should not happen because these are detected before)
        return True
    # ignore garbage and indentation at the beginning
    pos = 0
    while pos < len(ltok) and ltok[pos].type in [59, token.INDENT,
tokenize.ENCODING]:
        # 59 is BACKQUOTE but there is no token.BACKQUOTE...
        pos = pos + 1
    # when line is only garbage or does not begin with a name
    if pos >= len(ltok) or ltok[pos].type != token.NAME:
        return True
    command = ltok[pos].string
    if keyword.iskeyword(command):
        return True
    pos = pos + 1
    # command is alone on the line
    if pos >= len(ltok) or ltok[pos].type in [token.ENDMARKER,
tokenize.COMMENT]:
        if command in interpreter.locals:
            return True
        if interpreter.globals and command in interpreter.globals:
            return True
    else:  # command is not alone ; next token should not be an operator (this
includes parentheses)
        if ltok[pos].type == token.OP and not line.endswith("?"):
            return True
    return False
class Magician:
    def _eval(self, command):
        # Get namespace
        NS1 = sys._pyzoInterpreter.locals
        NS2 = sys._pyzoInterpreter.globals
        if not NS2:
            NS = NS1
        else:
            NS = NS2.copy()
            NS.update(NS1)
        # Evaluate in namespace
        return eval(command, {}, NS)
    def convert_command(self, line):
        """convert_command(line)
        Convert a given command from a magic command to Python code.
        Returns the converted command if it was a magic command, or
        the original otherwise.
        """
        # Get converted command, catch and report errors
        try:
            res = self._convert_command(line)
        except Exception:
            # Try informing about line number
            type, value, tb = sys.exc_info()
            msg = "Error in handling magic function:\n"
            msg += "  %s\n" % str(value)
            if tb.tb_next:
                tb = tb.tb_next.tb_next
            while tb:
                msg += "  line %i in %s\n" % (
                    tb.tb_lineno,
                    tb.tb_frame.f_code.co_filename,
                )
                tb = tb.tb_next
            # Clear
            del tb
            # Write
            print(msg)
            return None
        # Process
        if res is None:
            return line
        else:
            return res
    def _convert_command(self, line):
        # Get interpreter
        interpreter = sys._pyzoInterpreter
        command = line.rstrip()
        if interpreter._ipython and _detect_equalbang(line):
            return self.equalbang_quirk(line, command)
        have_hard_chars = "cd ", "?"
        if PYTHON_VERSION >= 3 and not
command.lower().startswith(have_hard_chars):
            try:
                if _should_not_interpret_as_magic(line):
                    return
            except Exception:
                pass  # dont break interpreter if above func has a bug ...
        else:
            # Old, not as good check for outdated Python version
            # Check if it is a variable
            if " " not in command:
                if command in interpreter.locals:
                    return
            if interpreter.globals and command in interpreter.globals:
                return
        # Clean and make case insensitive
        command = line.upper().rstrip()
        # Commands to always support (also with IPython)
        if not command:
            return
        elif command.startswith("DB"):
            return self.debug(line, command)
        elif command.startswith("NOTEBOOK"):
            return self.notebook(line, command)
        elif command.startswith("INSTALL"):
            return self.install(line, command)
        elif command.startswith("UPDATE") or command.startswith("UPGRADE"):
            line = line.replace("upgrade", "update")
            command = command.replace("UPGRADE", "UPDATE")
            return self.update(line, command)
        elif command.startswith("REMOVE") or command.startswith("UNINSTALL"):
            line = line.replace("uninstall", "remove")
            return self.remove(line, command)
        elif command.startswith("CONDA"):
            return self.conda(line, command)
        elif command.startswith("PIP"):
            return self.pip(line, command)
        elif command == "CLS":
            return self.cls(line, command)
        elif command.startswith("OPEN "):
            return self.open(line, command)
        elif interpreter._ipython:
            # Patch IPython magic
            # EDIT do not run code after editing
            if command.startswith("EDIT ") and " -X " not in command:
                return "edit -x " + line[5:]
            # In all cases stop processing magic commands
            return
        # Commands that we only support in the absense of IPtython
        elif command == "?":
            return "print(%s)" % repr(MESSAGE)
        elif command.startswith("??"):
            return "help(%s)" % line[2:].rstrip()
        elif command.endswith("??"):
            return "help(%s)" % line.rstrip()[:-2]
        elif command.startswith("?"):
            return "print(%s.__doc__)" % line[1:].rstrip()
        elif command.endswith("?"):
            return "print(%s.__doc__)" % line.rstrip()[:-1]
        elif command.startswith("CD"):
            return self.cd(line, command)
        elif command.startswith("LS"):
            return self.ls(line, command)
        elif command.startswith("TIMEIT"):
            return self.timeit(line, command)
        elif command == "WHO":
            return self.who(line, command)
        elif command == "WHOS":
            return self.whos(line, command)
        elif command.startswith("RUN "):
            return self.run(line, command)
    def debug(self, line, command):
        if command == "DB":
            line = "db help"
        elif not command.startswith("DB "):
            return
        # Get interpreter
        interpreter = sys._pyzoInterpreter
        # Get command and arg
        line += " "
        _, cmd, arg = line.split(" ", 2)
        cmd = cmd.lower()
        # Get func
        func = getattr(interpreter.debugger, "do_" + cmd, None)
        # Call or show warning
        if func is not None:
            func(arg.strip())
        else:
            interpreter.write("Unknown debug command: %s.\n" % cmd)
        # Done (no code to execute)
        return ""
    def equalbang_quirk(self, line, command):
        print("`=!' is not valid Python. To check for non-equality, use `!='.")
        print(
            "If you intend to use IPython's ! magic feature and assign its
result, please write `= !'."
        )
        return ""
    def cd(self, line, command):
        if command == "CD" or command.startswith("CD ") and "=" not in command:
            path = line[3:].strip()
            if path:
                try:
                    os.chdir(path)
                except Exception:
                    print('Could not change to directory "%s".' % path)
                    return ""
                newPath = os.getcwd()
            else:
                newPath = os.getcwd()
            # Done
            print(repr(newPath))
            return ""
    def ls(self, line, command):
        if command == "LS" or command.startswith("LS ") and "=" not in command:
            path = line[3:].strip()
            if not path:
                path = os.getcwd()
            L = [p for p in os.listdir(path) if not p.startswith(".")]
            text = "\n".join(sorted(L))
            # Done
            print(text)
            return ""
    def timeit(self, line, command):
        if command == "TIMEIT":
            return "print(%s)" % repr(TIMEIT_MESSAGE)
        elif command.startswith("TIMEIT "):
            expression = line[7:]
            # Get number of iterations
            N = 1
            tmp = expression.split(" ", 1)
            if len(tmp) == 2:
                try:
                    N = int(tmp[0])
                    expression = tmp[1]
                except Exception:
                    N = 1
            if expression[0] not in "'\"":
                isidentifier = lambda x: bool(re.match(r"[a-z_]\w*$", x, re.I))
                if not isidentifier(expression):
                    expression = "'%s'" % expression
            # Compile expression
            line2 = "import timeit; t=timeit.Timer(%s);" % expression
            line2 += "print(str( t.timeit(%i)/%i ) " % (N, N)
            line2 += '+" seconds on average for %i iterations." )' % N
            #
            return line2
    def who(self, line, command):
        L = self._eval("dir()\n")
        L = [k for k in L if not k.startswith("__")]
        if L:
            print(", ".join(L))
        else:
            print("There are no variables defined in this scope.")
        return ""
    def _justify(self, line, width, offset):
        realWidth = width - offset
        if len(line) > realWidth:
            line = line[: realWidth - 3] + "..."
        return line.ljust(width)
    def whos(self, line, command):
        # Get list of variables
        L = self._eval("dir()\n")
        L = [k for k in L if not k.startswith("__")]
        # Anny variables?
        if not L:
            print("There are no variables defined in this scope.")
            return ""
        else:
            text = "VARIABLE: ".ljust(20, " ") + "TYPE: ".ljust(20, " ")
            text += "REPRESENTATION: ".ljust(20, " ") + "\n"
        # Create list item for each variablee
        for name in L:
            ob = self._eval(name)
            cls = ""
            if hasattr(ob, "__class__"):
                cls = ob.__class__.__name__
            rep = repr(ob)
            text += self._justify(name, 20, 2) + self._justify(cls, 20, 2)
            text += self._justify(rep, 40, 2) + "\n"
        # Done
        print(text)
        return ""
    def cls(self, line, command):
        sys._pyzoInterpreter.context._strm_action.send("cls")
        return ""
    def open(self, line, command):
        # Get what to open
        name = line.split(" ", 1)[1].strip()
        fname = ""
        linenr = None
        # Is it a file name?
        tmp = os.path.join(os.getcwd(), name)
        #
        if name[0] in "\"'" and name[-1] in "\"'":  # Explicitly given
            fname = name[1:-1]
        elif os.path.isfile(tmp):
            fname = tmp
        elif os.path.isfile(name):
            fname = name
        else:
            # Then it maybe is an object
            # Get the object
            try:
                ob = self._eval(name)
            except NameError:
                print('There is no object known as "%s"' % name)
                return ""
            # Try get filename
            for iter in range(3):
                # Try successive steps
                if iter == 0:
                    ob = ob
                elif iter == 1 and not isinstance(ob, type):
                    ob = ob.__class__
                elif iter == 2 and hasattr(ob, "__module__"):
                    ob = sys.modules[ob.__module__]
                # Try get fname
                fname = ""
                try:
                    fname = inspect.getsourcefile(ob)
                except Exception:
                    pass
                # Returned fname may simply be x.replace('.pyc', '.py')
                if os.path.isfile(fname):
                    break
            # Try get line number
            if fname:
                try:
                    lines, linenr = inspect.getsourcelines(ob)
                except Exception:
                    pass
        # Almost done
        # IPython's edit now support this via our hook in interpreter.py
        if not fname:
            print('Could not determine file name for object "%s".' % name)
        elif linenr is not None:
            action = "open %i %s" % (linenr, os.path.abspath(fname))
            sys._pyzoInterpreter.context._strm_action.send(action)
        else:
            action = "open %s" % os.path.abspath(fname)
            sys._pyzoInterpreter.context._strm_action.send(action)
        #
        return ""
    def run(self, line, command):
        # Get what to open
        name = line.split(" ", 1)[1].strip()
        fname = ""
        # Enable dealing with qoutes
        if name[0] in "\"'" and name[-1] in "\"'":
            name = name[1:-1]
        # Is it a file name?
        tmp = os.path.join(os.getcwd(), name)
        #
        if os.path.isfile(tmp):
            fname = tmp
        elif os.path.isfile(name):
            fname = name
        # Go run!
        if not fname:
            print('Could not find file to run "%s".' % name)
        else:
            sys._pyzoInterpreter.runfile(fname)
        #
        return ""
    def _hasconda(self):
        try:
            from conda import __version__  # noqa
        except ImportError:
            return False
        return True
    def install(self, line, command):
        if not command.startswith("INSTALL "):
            return
        hasconda = self._hasconda()
        if hasconda:
            text = "Trying installation with conda. If this does not work,
try:\n"
            text += "   pip " + line + "\n"
        else:
            text = "Trying installation with pip\n"
        print("\x1b[34m\x1b[1m" + text + "\x1b[0m")
        time.sleep(0.2)
        if hasconda:
            self.conda("conda " + line, "CONDA")
        else:
            self.pip("pip " + line, "PIP")
        return ""
    def update(self, line, command):
        if not command.startswith("UPDATE "):
            return
        hasconda = self._hasconda()
        if hasconda:
            text = "Trying update with conda. If this does not work, try:\n"
            text += "   pip " + line.replace("update", "install") + "
--upgrade\n"
        else:
            text = "Trying pip install --upgrade\n"
        print("\x1b[34m\x1b[1m" + text + "\x1b[0m")
        time.sleep(0.2)
        if hasconda:
            self.conda("conda " + line, "CONDA")
        else:
            self.pip("pip " + line.replace("update", "install") + " --upgrade",
"PIP")
        return ""
    def remove(self, line, command):
        if not command.startswith(("REMOVE ", "UNINSTALL")):
            return
        hasconda = self._hasconda()
        if hasconda:
            text = "Trying remove/uninstall with conda. If this does not work,
try:\n"
            text += "   pip " + line.replace("remove", "uninstall") + "\n"
        else:
            text = "Trying remove/uninstall with pip\n"
        if hasconda:
            self.conda("conda " + line, "CONDA")
        else:
            self.pip("pip " + line.replace("remove", "uninstall"), "PIP")
        return ""
    def conda(self, line, command):
        if not (command == "CONDA" or command.startswith("CONDA ")):
            return
        # Get command args
        args = line.split(" ")
        args = [w for w in args if w]
        args.pop(0)  # remove 'conda'
        # Stop if user does "conda = ..."
        if args and "=" in args[0]:
            return
        # Add channels when using install, this gets added last, so
        # that user-specified channels take preference
        channel_list = []
        if args and args[0] == "install":
            channel_list = ["-c", "conda-forge", "-c", "pyzo"]
        def write_no_dots(x):
            if x.strip() == ".":  # note, no "x if y else z" in Python 2.4
                return 0
            return stderr_write(x)
        # Go!
        # Weird double-try, to make work on Python 2.4
        oldargs = sys.argv
        stderr_write = sys.stderr.write
        try:
            try:
                # older version of conda would spew dots to stderr during
downloading
                sys.stderr.write = write_no_dots
                import conda  # noqa
                from conda.cli import main
                sys.argv = ["conda"] + list(args) + channel_list
                main()
            except SystemExit:  # as err:
                type, err, tb = sys.exc_info()
                del tb
                err = str(err)
                if len(err) > 4:  # Only print if looks like a message
                    print(err)
            except Exception:  # as err:
                type, err, tb = sys.exc_info()
                del tb
                print("Error in conda command:")
                print(err)
        finally:
            sys.argv = oldargs
            sys.stderr.write = stderr_write
        return ""
    # todo: I think we can deprecate this
    def _check_imported_modules(self):
        KNOWN_PURE_PYHON = (
            "conda",
            "yaml",
            "IPython",
            "requests",
            "readline",
            "pyreadline",
        )
        if not sys.platform.startswith("win"):
            return  # Only a problem on Windows
        # Check what modules are currently imported
        loaded_modules = set()
        for name, mod in sys.modules.items():
            if "site-packages" in getattr(mod, "__file__", ""):
                name = name.split(".")[0]
                if name.startswith("_") or name in KNOWN_PURE_PYHON:
                    continue
                loaded_modules.add(name)
        # Add PySide PyQt4 from IEP if prefix is the same
        if os.getenv("IEP_PREFIX", "") == sys.prefix:
            loaded_modules.add(os.getenv("IEP_QTLIB", "qt"))
        # Warn if we have any such modules
        loaded_modules = [m.lower() for m in loaded_modules if m]
        if loaded_modules:
            print("WARNING! The following modules are currently loaded:\n")
            print("  " + ", ".join(sorted(loaded_modules)))
            print(
                "\nUpdating or removing them may fail if they are not "
                "pure Python.\nIf none of the listed packages is updated or "
                'removed, it is safe\nto proceed (use "f" if necessary).\n'
            )
            print("-" * 80)
            time.sleep(2.0)  # Give user time to realize there is a warning
    def pip(self, line, command):
        if not (command == "PIP" or command.startswith("PIP ")):
            return
        # Get command args
        args = line.split(" ")
        args = [w for w in args if w]
        args.pop(0)  # remove 'pip'
        # Stop if user does "pip = ..."
        if args and "=" in args[0]:
            return
        # Tweak the args
        if args and args[0] == "uninstall":
            args.insert(1, "--yes")
        # Go!
        try:
            from pyzokernel.pipper import pip_command
            pip_command(*args)
        except SystemExit:  # as err:
            type, err, tb = sys.exc_info()
            del tb
            err = str(err)
            if len(err) > 4:  # Only print if looks like a message
                print(err)
        except Exception:  # as err:
            type, err, tb = sys.exc_info()
            del tb
            print("Error in pip command:")
            print(err)
        return ""
    def notebook(self, line, command):
        if not (command == "NOTEBOOK" or command.startswith("NOTEBOOK ")):
            return
        # Get command args
        args = line.split(" ")
        args = [w.replace("%20", " ") for w in args if w]
        args.pop(0)  # remove 'notebook'
        # Stop if user does "conda = ..."
        if args and "=" in args[0]:
            return
        # Go!
        # Weird double-try, to make work on Python 2.4
        oldargs = sys.argv
        try:
            try:
                import notebook.notebookapp
                sys.argv = ["jupyter_notebook"] + list(args)
                notebook.notebookapp.main()
            except SystemExit:  # as err:
                type, err, tb = sys.exc_info()
                del tb
                err = str(err)
                if len(err) > 4:  # Only print if looks like a message
                    print(err)
            except Exception:  # as err:
                type, err, tb = sys.exc_info()
                del tb
                print("Error in notebook command:")
                print(err)
        finally:
            sys.argv = oldargs
        return ""
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, Almar Klein
# From pyzo/pyzokernel
import sys
import time
import subprocess
def subprocess_with_callback(callback, cmd, **kwargs):
    """Execute command in subprocess, stdout is passed to the
    callback function. Returns the returncode of the process.
    If callback is None, simply prints any stdout.
    """
    # Set callback to void if None
    if callback is None:
        callback = lambda x: None
    # Execute command
    try:
        p = subprocess.Popen(
            cmd, stdout=subprocess.PIPE, stderr=subprocess.STDOUT, **kwargs
        )
    except OSError:
        type, err, tb = sys.exc_info()
        del tb
        callback(str(err) + "\n")
        return -1
    pending = []
    while p.poll() is None:
        time.sleep(0.001)
        # Read text and process
        c = p.stdout.read(1).decode("utf-8", "ignore")
        pending.append(c)
        if c in ".\n":
            callback("".join(pending))
            pending = []
    # Process remaining text
    pending.append(p.stdout.read().decode("utf-8"))
    callback("".join(pending))
    # Done
    return p.returncode
def print_(p):
    sys.stdout.write(p)
    sys.stdout.flush()
def pip_command_exe(exe, *args):
    """Do a pip command in the interpreter with the given exe."""
    # Get pip command
    cmd = [exe, "-m", "pip"] + list(args)
    # Execute it
    subprocess_with_callback(print_, cmd)
def pip_command(*args):
    """Do a pip command, e.g. "install networkx".
    Installs in the current interpreter.
    """
    pip_command_exe(sys.executable, *args)
if __name__ == "__main__":
    pip_command("install", "networkx")
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" pyzokernel/start.py
Starting script for remote processes in pyzo.
This script connects to the pyzo ide using the yoton interface
and imports remote2 to start the interpreter and introspection thread.
Channels
--------
There are four groups of channels. The ctrl channels are streams from
the ide to the kernel and/or broker. The strm channels are streams to
the ide. The stat channels are status channels. The reqp channels are
req/rep channels. All channels are TEXT except for a few OBJECT channels.
ctrl-command: to give simple commands to the interpreter (ala stdin)
ctrl-code (OBJECT): to let the interpreter execute blocks of code
ctrl-broker: to control the broker (restarting etc)
strm-out: the stdout of the interpreter
strm-err: the stderr of the interpreter
strm-raw: the C-level stdout and stderr of the interpreter (captured by broker)
strm-echo: the interpreters echos commands here
strm-prompt: to send the prompts explicitly
strm-broker: for the broker to send messages to the ide
strm-action: for the kernel to push actions to the ide
stat-interpreter): status of the interpreter (ready, busy, very busy, more, etc)
stat-debug (OBJECT): debug status
stat-startup (OBJECT): Used to pass startup parameters to the kernel
reqp-introspect (OBJECT): To query information from the kernel (and for
interruping)
"""
# This file is executed with the active directory one up from this file.
import os
import sys
import time
import yoton
import __main__  # we will run code in the __main__.__dict__ namespace
## Make connection object and get channels
# Create a yoton context. All channels are stored at the context.
ct = yoton.Context()
# Create control channels
ct._ctrl_command = yoton.SubChannel(ct, "ctrl-command")
ct._ctrl_code = yoton.SubChannel(ct, "ctrl-code", yoton.OBJECT)
```

```
# Create stream channels
ct._strm_out = yoton.PubChannel(ct, "strm-out")
ct._strm_err = yoton.PubChannel(ct, "strm-err")
ct._strm_echo = yoton.PubChannel(ct, "strm-echo")
ct._strm_prompt = yoton.PubChannel(ct, "strm-prompt")
ct._strm_action = yoton.PubChannel(ct, "strm-action", yoton.OBJECT)
# Create status channels
ct._stat_interpreter = yoton.StateChannel(ct, "stat-interpreter")
ct._stat_cd = yoton.StateChannel(ct, "stat-cd")
ct._stat_debug = yoton.StateChannel(ct, "stat-debug", yoton.OBJECT)
ct._stat_startup = yoton.StateChannel(ct, "stat-startup", yoton.OBJECT)
ct._stat_breakpoints = yoton.StateChannel(ct, "stat-breakpoints", yoton.OBJECT)
# Connect (port number given as command line argument)
# Important to do this *before* replacing the stdout etc, because if an
# error occurs here, it will be printed in the shell.
port = int(sys.argv[1])
ct.connect("localhost:" + str(port), timeout=1.0)
# Create file objects for stdin, stdout, stderr
sys.stdin = yoton.FileWrapper(ct._ctrl_command, echo=ct._strm_echo, isatty=True)
sys.stdout = yoton.FileWrapper(ct._strm_out, 256, isatty=True)
sys.stderr = yoton.FileWrapper(ct._strm_err, 256, isatty=True)
# Set fileno on both
sys.stdout.fileno = sys.__stdout__.fileno
sys.stderr.fileno = sys.__stderr__.fileno
## Set Excepthook
def pyzo_excepthook(type, value, tb):
    import sys
    def writeErr(err):
        sys.__stderr__.write(str(err) + "\n")
        sys.__stderr__.flush()
    writeErr("Uncaught exception in interpreter:")
    writeErr(value)
    if not isinstance(value, (OverflowError, SyntaxError, ValueError)):
        while tb:
            writeErr(
                "-> line %i of %s."
                % (tb.tb_frame.f_lineno, tb.tb_frame.f_code.co_filename)
            )
            tb = tb.tb_next
    import time
    time.sleep(0.3)  # Give some time for the message to be send
# Uncomment to detect error in the interpreter itself.
# But better not use it by default. For instance errors in qt events
# are also catched by this function. I think that is because it would
# allow you to show such exceptions in an error dialog.
# sys.excepthook = pyzo_excepthook
## Init interpreter and introspector request channel
# Delay import, so we can detect syntax errors using the except hook
from pyzokernel.interpreter import PyzoInterpreter
from pyzokernel.introspection import PyzoIntrospector
# Create interpreter instance and give dict in which to run all code
__pyzo__ = PyzoInterpreter(__main__.__dict__, "<console>")
sys._pyzoInterpreter = __pyzo__
# Store context
__pyzo__.context = ct
# Create introspection req channel (store at interpreter instance)
__pyzo__.introspector = PyzoIntrospector(ct, "reqp-introspect")
## Clean up
# Delete local variables
del yoton, PyzoInterpreter, PyzoIntrospector, pyzo_excepthook
del ct, port
del os, sys, time
# Delete stuff we do not want
for name in [
    "__file__",  # __main__ does not have a corresponding file
    "__loader__",  # prevent lines from this file to be shown in tb
]:
    globals().pop(name, None)
del name
## Start and stop
# Start introspector and enter the interpreter
__pyzo__.introspector.set_mode("thread")
try:
    __pyzo__.run()
finally:
    # Restore original streams, so that SystemExit behaves as intended
    import sys
    try:
        sys.stdout, sys.stderr = sys.__stdout__, sys.__stderr__
    except Exception:
        pass
    # Flush pending messages (raises exception if times out)
    try:
        __pyzo__.context.flush(0.1)
    except Exception:
        pass
    # Nicely exit by closing context (closes channels and connections). If we do
    # not do this on Python 3.2 (at least Windows) the exit delays 10s. (issue
79)
    try:
        __pyzo__.introspector.set_mode(0)
        __pyzo__.context.close()
    except Exception:
        pass
```

```
# -----------------------------------------------------------------------------
#  Copyright (C) 2013 Min RK
#
#  Distributed under the terms of the 2-clause BSD License.
# -----------------------------------------------------------------------------
from contextlib import contextmanager
import ctypes
import ctypes.util
objc = ctypes.cdll.LoadLibrary(ctypes.util.find_library("objc"))
_ = ctypes.cdll.LoadLibrary(ctypes.util.find_library("Foundation"))
void_p = ctypes.c_void_p
ull = ctypes.c_uint64
objc.objc_getClass.restype = void_p
objc.sel_registerName.restype = void_p
objc.objc_msgSend.restype = void_p
objc.objc_msgSend.argtypes = [void_p, void_p]
msg = objc.objc_msgSend
def _utf8(s):
    """ensure utf8 bytes"""
    if not isinstance(s, bytes):
        s = s.encode("utf8")
    return s
def n(name):
    """create a selector name (for methods)"""
    return objc.sel_registerName(_utf8(name))
def C(classname):
    """get an ObjC Class by name"""
    ret = objc.objc_getClass(_utf8(classname))
    assert ret is not None, "Couldn't find Class %s" % classname
    return ret
# constants from Foundation
NSActivityIdleDisplaySleepDisabled = 1 << 40
NSActivityIdleSystemSleepDisabled = 1 << 20
NSActivitySuddenTerminationDisabled = 1 << 14
NSActivityAutomaticTerminationDisabled = 1 << 15
NSActivityUserInitiated = 0x00FFFFFF | NSActivityIdleSystemSleepDisabled
NSActivityUserInitiatedAllowingIdleSystemSleep = (
    NSActivityUserInitiated & ~NSActivityIdleSystemSleepDisabled
)
NSActivityBackground = 0x000000FF
NSActivityLatencyCritical = 0xFF00000000
def beginActivityWithOptions(options, reason=""):
    """Wrapper for:
```

```
    [ [ NSProcessInfo processInfo]
        beginActivityWithOptions: (uint64)options
                          reason: (str)reason
    ]
    """
    NSProcessInfo = C("NSProcessInfo")
    NSString = C("NSString")
    objc.objc_msgSend.argtypes = [void_p, void_p, void_p]
    reason = msg(NSString, n("stringWithUTF8String:"), _utf8(reason))
    objc.objc_msgSend.argtypes = [void_p, void_p]
    info = msg(NSProcessInfo, n("processInfo"))
    objc.objc_msgSend.argtypes = [void_p, void_p, ull, void_p]
    activity = msg(
        info, n("beginActivityWithOptions:reason:"), ull(options),
void_p(reason)
    )
    return activity
def endActivity(activity):
    """end a process activity assertion"""
    NSProcessInfo = C("NSProcessInfo")
    objc.objc_msgSend.argtypes = [void_p, void_p]
    info = msg(NSProcessInfo, n("processInfo"))
    objc.objc_msgSend.argtypes = [void_p, void_p, void_p]
    msg(info, n("endActivity:"), void_p(activity))
_theactivity = None
def nope():
    """disable App Nap by setting
NSActivityUserInitiatedAllowingIdleSystemSleep"""
    global _theactivity
    _theactivity = beginActivityWithOptions(
        NSActivityUserInitiatedAllowingIdleSystemSleep, "Because Reasons"
    )
def nap():
    """end the caffeinated state started by `nope`"""
    global _theactivity
    if _theactivity is not None:
        endActivity(_theactivity)
        _theactivity = None
def napping_allowed():
    """is napping allowed?"""
    return _theactivity is None
@contextmanager
def nope_scope(
    options=NSActivityUserInitiatedAllowingIdleSystemSleep, reason="Because
Reasons"
):
    """context manager for beginActivityWithOptions.
    Within this context, App Nap will be disabled.
    """
    activity = beginActivityWithOptions(options, reason)
    try:
        yield
    finally:
        endActivity(activity)
__all__ = [
    "NSActivityIdleDisplaySleepDisabled",
    "NSActivityIdleSystemSleepDisabled",
    "NSActivitySuddenTerminationDisabled",
    "NSActivityAutomaticTerminationDisabled",
    "NSActivityUserInitiated",
    "NSActivityUserInitiatedAllowingIdleSystemSleep",
    "NSActivityBackground",
    "NSActivityLatencyCritical",
    "beginActivityWithOptions",
    "endActivity",
    "nope",
    "nap",
    "napping_allowed",
    "nope_scope",
]
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
"""
The pyzokernel package contains the code for the Pyzo kernel process.
This kernel is designed to be relatively lightweight; i.e. most of
the work is done by the IDE.
See pyzokernel/start.py for more information.
"""
def printDirect(msg):
    """Small function that writes directly to the strm_out channel.
    This means that regardless if stdout was hijacked, the message ends
    up at the Pyzo shell. This keeps any hijacked stdout clean, and gets
    the message where you want it. In most cases this is just cosmetics:
    the Python banner is one example.
    """
    import sys
    sys._pyzoInterpreter.context._strm_out.send(msg)
```

```
#
# Copyright © 2009- The Spyder Development Team
# Licensed under the terms of the MIT License
"""
Compatibility functions
"""
import sys
from .QtWidgets import QFileDialog
TEXT_TYPES = (str,)
def is_text_string(obj):
    """Return True if `obj` is a text string, False if it is anything else,
    like binary data."""
    return isinstance(obj, str)
def to_text_string(obj, encoding=None):
    """Convert `obj` to (unicode) text string"""
    if encoding is None:
        return str(obj)
    elif isinstance(obj, str):
        # In case this function is not used properly, this could happen
        return obj
    else:
        return str(obj, encoding)
# =============================================================================
# QVariant conversion utilities
# =============================================================================
PYQT_API_1 = False
def to_qvariant(obj=None):  # analysis:ignore
    """Convert Python object to QVariant
    This is a transitional function from PyQt API#1 (QVariant exist)
    to PyQt API#2 and Pyside (QVariant does not exist)"""
    return obj
def from_qvariant(qobj=None, pytype=None):  # analysis:ignore
    """Convert QVariant object to Python object
    This is a transitional function from PyQt API #1 (QVariant exist)
    to PyQt API #2 and Pyside (QVariant does not exist)"""
    return qobj
# =============================================================================
# Wrappers around QFileDialog static methods
# =============================================================================
def getexistingdirectory(
    parent=None, caption="", basedir="", options=QFileDialog.ShowDirsOnly
):
    """Wrapper around QtGui.QFileDialog.getExistingDirectory static method
    Compatible with PyQt >=v4.4 (API #1 and #2) and PySide >=v1.0"""
    # Calling QFileDialog static method
    if sys.platform == "win32":
        # On Windows platforms: redirect standard outputs
        _temp1, _temp2 = sys.stdout, sys.stderr
        sys.stdout, sys.stderr = None, None
    try:
        result = QFileDialog.getExistingDirectory(parent, caption, basedir,
options)
    finally:
        if sys.platform == "win32":
            # On Windows platforms: restore standard outputs
            sys.stdout, sys.stderr = _temp1, _temp2
    if not is_text_string(result):
        # PyQt API #1
        result = to_text_string(result)
    return result
def _qfiledialog_wrapper(
    attr,
    parent=None,
    caption="",
    basedir="",
    filters="",
    selectedfilter="",
    options=None,
):
    if options is None:
        options = QFileDialog.Options(0)
    func = getattr(QFileDialog, attr)
    # Calling QFileDialog static method
    if sys.platform == "win32":
        # On Windows platforms: redirect standard outputs
        _temp1, _temp2 = sys.stdout, sys.stderr
        sys.stdout, sys.stderr = None, None
    result = func(parent, caption, basedir, filters, selectedfilter, options)
    if sys.platform == "win32":
        # On Windows platforms: restore standard outputs
        sys.stdout, sys.stderr = _temp1, _temp2
    output, selectedfilter = result
    # Always returns the tuple (output, selectedfilter)
    return output, selectedfilter
def getopenfilename(
    parent=None, caption="", basedir="", filters="", selectedfilter="",
options=None
):
    """Wrapper around QtGui.QFileDialog.getOpenFileName static method
    Returns a tuple (filename, selectedfilter) -- when dialog box is canceled,
    returns a tuple of empty strings
    Compatible with PyQt >=v4.4 (API #1 and #2) and PySide >=v1.0"""
    return _qfiledialog_wrapper(
        "getOpenFileName",
        parent=parent,
        caption=caption,
        basedir=basedir,
        filters=filters,
        selectedfilter=selectedfilter,
        options=options,
    )
def getopenfilenames(
    parent=None, caption="", basedir="", filters="", selectedfilter="",
options=None
):
    """Wrapper around QtGui.QFileDialog.getOpenFileNames static method
    Returns a tuple (filenames, selectedfilter) -- when dialog box is canceled,
    returns a tuple (empty list, empty string)
    Compatible with PyQt >=v4.4 (API #1 and #2) and PySide >=v1.0"""
    return _qfiledialog_wrapper(
        "getOpenFileNames",
        parent=parent,
        caption=caption,
        basedir=basedir,
        filters=filters,
        selectedfilter=selectedfilter,
        options=options,
    )
def getsavefilename(
    parent=None, caption="", basedir="", filters="", selectedfilter="",
options=None
):
    """Wrapper around QtGui.QFileDialog.getSaveFileName static method
    Returns a tuple (filename, selectedfilter) -- when dialog box is canceled,
    returns a tuple of empty strings
    Compatible with PyQt >=v4.4 (API #1 and #2) and PySide >=v1.0"""
    return _qfiledialog_wrapper(
        "getSaveFileName",
        parent=parent,
        caption=caption,
        basedir=basedir,
        filters=filters,
        selectedfilter=selectedfilter,
        options=options,
    )
```

```
# Copyright © 2009- The Spyder Development Team
# Copyright © 2012- University of North Carolina at Chapel Hill
#                   Luke Campagnola    ('luke.campagnola@%s.com' % 'gmail')
#                   Ogi Moore          ('ognyan.moore@%s.com' % 'gmail')
#                   KIU Shueng Chuan   ('nixchuan@%s.com' % 'gmail')
# Licensed under the terms of the MIT License
"""
Compatibility functions for scoped and unscoped enum access.
"""
from . import PYQT6, PYSIDE6
if PYQT6:
    import enum
    from . import sip
    def promote_enums(module):
        """
        Search enums in the given module and allow unscoped access.
        Taken from:
        https://github.com/pyqtgraph/pyqtgraph/blob/pyqtgraph-
0.12.1/pyqtgraph/Qt.py#L331-L377
        """
        class_names = [name for name in dir(module) if name.startswith("Q")]
        for class_name in class_names:
            klass = getattr(module, class_name)
            if not isinstance(klass, sip.wrappertype):
                continue
            attrib_names = [name for name in dir(klass) if name[0].isupper()]
            for attrib_name in attrib_names:
                attrib = getattr(klass, attrib_name)
                if not isinstance(attrib, enum.EnumMeta):
                    continue
                for enum_obj_name in attrib.__members__:
                    setattr(klass, enum_obj_name, attrib[enum_obj_name])
elif PYSIDE6:
    from enum import Enum
    def promote_enums(module):
        for ob in (getattr(module, name) for name in dir(module)):
            if isinstance(ob, type) and ob.__name__.startswith("Q"):
                fix_enums(ob)
    def new_getattr(self, key):
        try:
            return self.__ori_getattr__(key)
        except AttributeError:
            pass
        return getattr(self.__class__, key)
    def stub_getattr(self, name):
        raise AttributeError()
    def fix_enums(cls):
        ori = getattr(cls, "__getattr__", stub_getattr)
        if ori is not new_getattr:
            cls.__ori_getattr__ = ori
            cls.__getattr__ = new_getattr
```

```
#
# Copyright © 2014-2015 Colin Duquesnoy
# Copyright © 2009- The Spyder Development Team
#
# Licensed under the terms of the MIT License
# (see LICENSE.txt for details)
"""
Provides QtCore classes and functions.
"""
from . import PYQT6, PYQT5, PYSIDE2, PYSIDE6, PythonQtError
if PYQT6:
    from PyQt6 import QtCore
    from PyQt6.QtCore import *
    from PyQt6.QtCore import pyqtSignal as Signal
    from PyQt6.QtCore import pyqtBoundSignal as SignalInstance
    from PyQt6.QtCore import pyqtSlot as Slot
    from PyQt6.QtCore import pyqtProperty as Property
    from PyQt6.QtCore import QT_VERSION_STR as __version__
    # For issue #153
    from PyQt6.QtCore import QDateTime
    QDateTime.toPython = QDateTime.toPyDateTime
    # Map missing methods
    QCoreApplication.exec_ = QCoreApplication.exec
    QEventLoop.exec_ = QEventLoop.exec
    QThread.exec_ = QThread.exec
    QLibraryInfo.location = QLibraryInfo.path
    # Those are imported from `import *`
    del pyqtSignal, pyqtBoundSignal, pyqtSlot, pyqtProperty, QT_VERSION_STR
    from .enums_compat import promote_enums
    promote_enums(QtCore)
    QtCore.Qt.MidButton = QtCore.Qt.MiddleButton
    del QtCore
elif PYQT5:
    from PyQt5.QtCore import *
    from PyQt5.QtCore import pyqtSignal as Signal
    from PyQt5.QtCore import pyqtBoundSignal as SignalInstance
    from PyQt5.QtCore import pyqtSlot as Slot
    from PyQt5.QtCore import pyqtProperty as Property
    from PyQt5.QtCore import QT_VERSION_STR as __version__
    # For issue #153
    from PyQt5.QtCore import QDateTime
    QDateTime.toPython = QDateTime.toPyDateTime
    # Those are imported from `import *`
    del pyqtSignal, pyqtBoundSignal, pyqtSlot, pyqtProperty, QT_VERSION_STR
elif PYSIDE6:
    from PySide6.QtCore import *
    import PySide6
    from PySide6 import QtCore
    __version__ = PySide6.QtCore.__version__
    # Map DeprecationWarning methods
    QCoreApplication.exec_ = QCoreApplication.exec
    QEventLoop.exec_ = QEventLoop.exec
    QThread.exec_ = QThread.exec
    QTextStreamManipulator.exec_ = QTextStreamManipulator.exec
    from .enums_compat import promote_enums
    promote_enums(QtCore)
    QtCore.Qt.MidButton = QtCore.Qt.MiddleButton
    del QtCore
elif PYSIDE2:
    from PySide2.QtCore import *
    try:  # may be limited to PySide-5.11a1 only
        from PySide2.QtGui import QStringListModel
    except Exception:
        pass
    import PySide2.QtCore
    __version__ = PySide2.QtCore.__version__
else:
    raise PythonQtError("No Qt bindings could be found")
```

```
#
# Copyright © 2014-2015 Colin Duquesnoy
# Copyright © 2009- The Spyder Development Team
#
# Licensed under the terms of the MIT License
# (see LICENSE.txt for details)
"""
Provides QtGui classes and functions.
"""
from . import PYQT6, PYQT5, PYSIDE2, PYSIDE6, PythonQtError
if PYQT6:
    from PyQt6.QtGui import *
    import inspect
    QFontMetrics.width = inspect.getattr_static(QFontMetrics,
"horizontalAdvance")
    # Map missing/renamed methods
    QDrag.exec_ = inspect.getattr_static(QDrag, "exec")
    QGuiApplication.exec_ = inspect.getattr_static(QGuiApplication, "exec")
    QTextDocument.print_ = inspect.getattr_static(QTextDocument, "print")
    QTextDocument.FindFlags = lambda: QTextDocument.FindFlag(0)
    from .enums_compat import promote_enums
    from PyQt6 import QtGui
    QtGui.QMouseEvent.x = lambda self: int(self.position().x())
    QtGui.QMouseEvent.y = lambda self: int(self.position().y())
    if not hasattr(QtGui.QMouseEvent, "pos"):
        QtGui.QMouseEvent.pos = lambda self: self.position().toPoint()
    promote_enums(QtGui)
    # in Qt6 use QtGui.QFontDatabase.families() instead of
QtGui.QFontDatabase().families()
    # https://doc.qt.io/qt-6/qfontdatabase-obsolete.html#QFontDatabase
    QtGui.QFontDatabase.__new__ = lambda cls: cls
    del QtGui
    del inspect
elif PYQT5:
    from PyQt5.QtGui import *
elif PYSIDE2:
    from PySide2.QtGui import *
elif PYSIDE6:
    from PySide6 import QtGui
    from PySide6.QtGui import *
    QFontMetrics.width = QFontMetrics.horizontalAdvance
    # Map DeprecationWarning methods
    QDrag.exec_ = QDrag.exec
    QGuiApplication.exec_ = QGuiApplication.exec
    from .enums_compat import promote_enums
    promote_enums(QtGui)
else:
    raise PythonQtError("No Qt bindings could be found")
```

```
#
# Copyright © 2009- The Spyder Development Team
#
# Licensed under the terms of the MIT License
# (see LICENSE.txt for details)
"""QtHelp Wrapper."""
import warnings
from . import PYQT5, PYQT6, PYSIDE6, PYSIDE2, PythonQtError
if PYQT5:
    from PyQt5.QtHelp import *
elif PYQT6:
    from PyQt6.QtHelp import *
elif PYSIDE6:
    from PySide6.QtHelp import *
elif PYSIDE2:
    from PySide2.QtHelp import *
else:
    raise PythonQtError("No Qt bindings could be found")
```

```
#
# Copyright © 2009- The Spyder Development Team
#
# Licensed under the terms of the MIT License
# (see LICENSE.txt for details)
"""
Provides QtPrintSupport classes and functions.
"""
from . import PYQT5, PYQT6, PYSIDE6, PYSIDE2, PythonQtError
if PYQT5:
    from PyQt5.QtPrintSupport import *
elif PYQT6:
    from PyQt6.QtPrintSupport import *
    QPageSetupDialog.exec_ = QPageSetupDialog.exec
    QPrintDialog.exec_ = QPrintDialog.exec
    QPrintPreviewWidget.print_ = QPrintPreviewWidget.print
    from .enums_compat import promote_enums
    from PyQt6 import QtPrintSupport
    promote_enums(QtPrintSupport)
    del QtPrintSupport
elif PYSIDE6:
    from PySide6.QtPrintSupport import *
    # Map DeprecationWarning methods
    QPageSetupDialog.exec_ = QPageSetupDialog.exec
    QPrintDialog.exec_ = QPrintDialog.exec
elif PYSIDE2:
    from PySide2.QtPrintSupport import *
else:
    raise PythonQtError("No Qt bindings could be found")
```

```
#
# Copyright © 2014-2015 Colin Duquesnoy
# Copyright © 2009- The Spyder Developmet Team
#
# Licensed under the terms of the MIT License
# (see LICENSE.txt for details)
"""
Provides widget classes and functions.
"""
from . import PYQT5, PYQT6, PYSIDE2, PYSIDE6, PythonQtError
if PYQT6:
    from PyQt6.QtWidgets import *
    from PyQt6.QtGui import QAction, QActionGroup, QShortcut, QFileSystemModel
    from PyQt6.QtOpenGLWidgets import QOpenGLWidget
    # Map missing/renamed methods
    import inspect
    QTextEdit.setTabStopWidth = inspect.getattr_static(QTextEdit,
"setTabStopDistance")
    QTextEdit.tabStopWidth = inspect.getattr_static(QTextEdit,
"tabStopDistance")
    QTextEdit.print_ = inspect.getattr_static(QTextEdit, "print")
    QPlainTextEdit.setTabStopWidth = inspect.getattr_static(QPlainTextEdit,
"setTabStopDistance")
    QPlainTextEdit.tabStopWidth = inspect.getattr_static(QPlainTextEdit,
"tabStopDistance")
    QPlainTextEdit.print_ = inspect.getattr_static(QPlainTextEdit, "print")
    if QApplication.__name__ != "QApplication_hijacked":
        QApplication.exec_ = inspect.getattr_static(QApplication, "exec")
    else:
        print("Pyzo is executed inside another Pyzo instance")
    QDialog.exec_ = inspect.getattr_static(QDialog, "exec")
    QMenu.exec_ = inspect.getattr_static(QMenu, "exec")
    from PyQt6 import QtWidgets
    from .enums_compat import promote_enums
    promote_enums(QtWidgets)
    del QtWidgets
    del inspect
elif PYQT5:
    from PyQt5.QtWidgets import *
elif PYSIDE6:
    from PySide6.QtWidgets import *
    from PySide6.QtGui import QAction, QActionGroup, QShortcut
    from PySide6.QtOpenGLWidgets import QOpenGLWidget
    # Map missing/renamed methods
    QTextEdit.setTabStopWidth = QTextEdit.setTabStopDistance
    QTextEdit.tabStopWidth = QTextEdit.tabStopDistance
    QPlainTextEdit.setTabStopWidth = QPlainTextEdit.setTabStopDistance
    QPlainTextEdit.tabStopWidth = QPlainTextEdit.tabStopDistance
    # Map DeprecationWarning methods
    if QApplication.__name__ != "QApplication_hijacked":
        QApplication.exec_ = QApplication.exec
    else:
        print("Pyzo is executed inside another Pyzo instance")
    QDialog.exec_ = QDialog.exec
    QMenu.exec_ = QMenu.exec
    from PySide6 import QtWidgets
    from .enums_compat import promote_enums
    promote_enums(QtWidgets)
    del QtWidgets
elif PYSIDE2:
    from PySide2.QtWidgets import *
else:
    raise PythonQtError("No Qt bindings could be found")
```

```
#
# Copyright © 2009- The Spyder Development Team
#
# Licensed under the terms of the MIT License
# (see LICENSE.txt for details)
from . import PYQT6, PYQT5, PythonQtError
if PYQT6:
    from PyQt6.sip import *
elif PYQT5:
    from PyQt5.sip import *
else:
    raise PythonQtError("Currently selected Qt binding does not support this module")
```

```
import os
from . import PYSIDE6, PYSIDE2, PYQT5, PYQT6
from .QtWidgets import QComboBox
if PYQT6:
    from PyQt6.uic import *
elif PYQT5:
    from PyQt5.uic import *
else:
    __all__ = ["loadUi", "loadUiType"]
    # In PySide, loadUi does not exist, so we define it using QUiLoader, and
    # then make sure we expose that function. This is adapted from qt-helpers
    # which was released under a 3-clause BSD license:
    # qt-helpers - a common front-end to various Qt modules
    #
    # Copyright (c) 2015, Chris Beaumont and Thomas Robitaille
    #
    # All rights reserved.
    #
    # Redistribution and use in source and binary forms, with or without
    # modification, are permitted provided that the following conditions are
    # met:
    #
    #  * Redistributions of source code must retain the above copyright
    #    notice, this list of conditions and the following disclaimer.
    #  * Redistributions in binary form must reproduce the above copyright
    #    notice, this list of conditions and the following disclaimer in the
    #    documentation and/or other materials provided with the
    #    distribution.
    #  * Neither the name of the Glue project nor the names of its contributors
    #    may be used to endorse or promote products derived from this software
    #    without specific prior written permission.
    #
    # THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
    # IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
    # THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    # PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
    # CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
    # EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
    # PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
    # PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
    # LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
    # NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
    # SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

```
#
# Which itself was based on the solution at
#
# https://gist.github.com/cpbotha/1b42a20c8f3eb9bb7cb8
#
# which was released under the MIT license:
#
# Copyright (c) 2011 Sebastian Wiesner <lunaryorn@gmail.com>
# Modifications by Charl Botha <cpbotha@vxlabs.com>
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the "Software"),
# to deal in the Software without restriction, including without limitation
# the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the
# Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
# FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
# DEALINGS IN THE SOFTWARE.
if PYSIDE6:
    from PySide6.QtCore import QMetaObject
    from PySide6.QtUiTools import QUiLoader
elif PYSIDE2:
    from PySide2.QtCore import QMetaObject
    from PySide2.QtUiTools import QUiLoader
    try:
        from pyside2uic import compileUi
    except ImportError:
        pass
class UiLoader(QUiLoader):
    """
    Subclass of :class:`~PySide.QtUiTools.QUiLoader` to create the user
    interface in a base instance.
    Unlike :class:`~PySide.QtUiTools.QUiLoader` itself this class does not
    create a new instance of the top-level widget, but creates the user
```

```
        interface in an existing instance of the top-level class if needed.
        This mimics the behaviour of :func:`PyQt4.uic.loadUi`.
        """
        def __init__(self, baseinstance, customWidgets=None):
            """
            Create a loader for the given ``baseinstance``.
            The user interface is created in ``baseinstance``, which must be an
            instance of the top-level class in the user interface to load, or a
            subclass thereof.
            ``customWidgets`` is a dictionary mapping from class name to class
            object for custom widgets. Usually, this should be done by calling
            registerCustomWidget on the QUiLoader, but with PySide 1.1.2 on
            Ubuntu 12.04 x86_64 this causes a segfault.
            ``parent`` is the parent object of this loader.
            """
            QUiLoader.__init__(self, baseinstance)
            self.baseinstance = baseinstance
            if customWidgets is None:
                self.customWidgets = {}
            else:
                self.customWidgets = customWidgets
        def createWidget(self, class_name, parent=None, name=""):
            """
            Function that is called for each widget defined in ui file,
            overridden here to populate baseinstance instead.
            """
            if parent is None and self.baseinstance:
                # supposed to create the top-level widget, return the base
                # instance instead
                return self.baseinstance
            else:
                # For some reason, Line is not in the list of available
                # widgets, but works fine, so we have to special case it here.
                if class_name in self.availableWidgets() or class_name ==
"Line":
                    # create a new widget for child widgets
                    widget = QUiLoader.createWidget(self, class_name, parent,
name)
                else:
                    # If not in the list of availableWidgets, must be a custom
                    # widget. This will raise KeyError if the user has not
                    # supplied the relevant class_name in the dictionary or if
                    # customWidgets is empty.
                    try:
                        widget = self.customWidgets[class_name](parent)
                    except KeyError as error:
                        raise Exception(
                            f"No custom widget {class_name} " "found in
customWidgets"
                        ) from error
                if self.baseinstance:
                    # set an attribute for the new child widget on the base
                    # instance, just like PyQt4.uic.loadUi does.
                    setattr(self.baseinstance, name, widget)
                return widget
    def _get_custom_widgets(ui_file):
        """
        This function is used to parse a ui file and look for the
<customwidgets>
        section, then automatically load all the custom widget classes.
        """
        import sys
        import importlib
        from xml.etree.ElementTree import ElementTree
        # Parse the UI file
        etree = ElementTree()
        ui = etree.parse(ui_file)
        # Get the customwidgets section
        custom_widgets = ui.find("customwidgets")
        if custom_widgets is None:
            return {}
        custom_widget_classes = {}
        for custom_widget in list(custom_widgets):
            cw_class = custom_widget.find("class").text
            cw_header = custom_widget.find("header").text
            module = importlib.import_module(cw_header)
            custom_widget_classes[cw_class] = getattr(module, cw_class)
        return custom_widget_classes
    def loadUi(uifile, baseinstance=None, workingDirectory=None):
        """
        Dynamically load a user interface from the given ``uifile``.
        ``uifile`` is a string containing a file name of the UI file to load.
        If ``baseinstance`` is ``None``, the a new instance of the top-level
        widget will be created. Otherwise, the user interface is created within
        the given ``baseinstance``. In this case ``baseinstance`` must be an
        instance of the top-level widget class in the UI file to load, or a
        subclass thereof. In other words, if you've created a ``QMainWindow``
        interface in the designer, ``baseinstance`` must be a ``QMainWindow``
        or a subclass thereof, too. You cannot load a ``QMainWindow`` UI file
        with a plain :class:`~PySide.QtGui.QWidget` as ``baseinstance``.
        :method:`~PySide.QtCore.QMetaObject.connectSlotsByName()` is called on
        the created user interface, so you can implemented your slots according
        to its conventions in your widget class.
        Return ``baseinstance``, if ``baseinstance`` is not ``None``. Otherwise
        return the newly created instance of the user interface.
        """
        # We parse the UI file and import any required custom widgets
        customWidgets = _get_custom_widgets(uifile)
        loader = UiLoader(baseinstance, customWidgets)
        if workingDirectory is not None:
            loader.setWorkingDirectory(workingDirectory)
        widget = loader.load(uifile)
        QMetaObject.connectSlotsByName(widget)
        return widget
    def loadUiType(uifile, from_imports=False):
        """Load a .ui file and return the generated form class and
        the Qt base class.
        The "loadUiType" command convert the ui file to py code
        in-memory first and then execute it in a special frame to
        retrieve the form_class.
        Credit: https://stackoverflow.com/a/14195313/15954282
        """
        import sys
        from io import StringIO
        from xml.etree.ElementTree import ElementTree
        from . import QtWidgets
        # Parse the UI file
        etree = ElementTree()
        ui = etree.parse(uifile)
        widget_class = ui.find("widget").get("class")
        form_class = ui.find("class").text
        with open(uifile, encoding="utf-8") as fd:
            code_stream = StringIO()
            frame = {}
            compileUi(fd, code_stream, indent=0, from_imports=from_imports)
            pyc = compile(code_stream.getvalue(), "<string>", "exec")
            exec(pyc, frame)
            # Fetch the base_class and form class based on their type in the
            # xml from designer
```

```
            form_class = frame["Ui_%s" % form_class]
            base_class = getattr(QtWidgets, widget_class)
        return form_class, base_class
```

```
#
# Copyright © 2009- The Spyder Development Team
# Copyright © 2014-2015 Colin Duquesnoy
#
# Licensed under the terms of the MIT License
# (see LICENSE.txt for details)
"""
**QtPy** is a shim over the various Python Qt bindings. It is used to write
Qt binding independent libraries or applications.
If one of the APIs has already been imported, then it will be used.
Otherwise, the shim will automatically select the first available API (PyQt5,
PyQt6,
PySide2 and PySide6); in that case, you can force the use of one
specific bindings (e.g. if your application is using one specific bindings and
you need to use library that use QtPy) by setting up the ``QT_API`` environment
variable.
PyQt5
=====
For PyQt5, you don't have to set anything as it will be used automatically::
    >>> from qtpy import QtGui, QtWidgets, QtCore
    >>> print(QtWidgets.QWidget)
PyQt6
=====
    >>> import os
    >>> os.environ['QT_API'] = 'pyqt6'
    >>> from qtpy import QtGui, QtWidgets, QtCore
    >>> print(QtWidgets.QWidget)
PySide2
======
Set the QT_API environment variable to 'pyside2' before importing other
packages::
    >>> import os
    >>> os.environ['QT_API'] = 'pyside2'
    >>> from qtpy import QtGui, QtWidgets, QtCore
    >>> print(QtWidgets.QWidget)
PySide6
=======
    >>> import os
    >>> os.environ['QT_API'] = 'pyside6'
    >>> from qtpy import QtGui, QtWidgets, QtCore
    >>> print(QtWidgets.QWidget)
"""
from packaging.version import parse
```

```
import os
import platform
import sys
import warnings
# Version of QtPy
__version__ = "2.0.0.dev0"
class PythonQtError(RuntimeError):
    """Error raised if no bindings could be selected."""
class PythonQtWarning(Warning):
    """Warning if some features are not implemented in a binding."""
# Qt API environment variable name
QT_API = "QT_API"
# Names of the expected PyQt5 api
PYQT5_API = ["pyqt5"]
PYQT6_API = ["pyqt6"]
# Names of the expected PySide2 api
PYSIDE2_API = ["pyside2"]
# Names of the expected PySide6 api
PYSIDE6_API = ["pyside6"]
# Minimum supported versions of Qt and the bindings
QT5_VERSION_MIN = PYQT5_VERSION_MIN = "5.9.0"
PYSIDE2_VERSION_MIN = "5.12.0"
QT6_VERSION_MIN = PYQT6_VERSION_MIN = PYSIDE6_VERSION_MIN = "6.2.0"
QT_VERSION_MIN = QT5_VERSION_MIN
PYQT_VERSION_MIN = PYQT5_VERSION_MIN
PYSIDE_VERISION_MIN = PYSIDE2_VERSION_MIN
# Detecting if a binding was specified by the user
binding_specified = QT_API in os.environ
# Setting a default value for QT_API
os.environ.setdefault(QT_API, "pyqt5")
API = os.environ[QT_API].lower()
initial_api = API
assert API in (PYQT5_API + PYQT6_API + PYSIDE2_API + PYSIDE6_API)
is_old_pyqt = is_pyqt46 = False
QT5 = PYQT5 = True
QT4 = QT6 = PYQT4 = PYQT6 = PYSIDE = PYSIDE2 = PYSIDE6 = False
PYQT_VERSION = None
PYSIDE_VERSION = None
QT_VERSION = None
# Unless `FORCE_QT_API` is set, use previously imported Qt Python bindings
if not os.environ.get("FORCE_QT_API"):
    if "PyQt6" in sys.modules:
        API = initial_api if initial_api in PYQT6_API else "pyqt6"
    elif "PyQt5" in sys.modules:
        API = initial_api if initial_api in PYQT5_API else "pyqt5"
    elif "PySide6" in sys.modules:
        API = initial_api if initial_api in PYSIDE6_API else "pyside6"
    elif "PySide2" in sys.modules:
        API = initial_api if initial_api in PYSIDE2_API else "pyside2"
if API in PYQT5_API:
    try:
        from PyQt5.QtCore import PYQT_VERSION_STR as PYQT_VERSION  #
analysis:ignore
        from PyQt5.QtCore import QT_VERSION_STR as QT_VERSION  # analysis:ignore
        QT5 = PYQT5 = True
        if sys.platform == "darwin":
            macos_version = parse(platform.mac_ver()[0])
            if macos_version < parse("10.10"):
                if parse(QT_VERSION) >= parse("5.9"):
                    raise PythonQtError(
                        "Qt 5.9 or higher only works in "
                        "macOS 10.10 or higher. Your "
                        "program will fail in this "
                        "system."
                    )
            elif macos_version < parse("10.11"):
                if parse(QT_VERSION) >= parse("5.11"):
                    raise PythonQtError(
                        "Qt 5.11 or higher only works in "
                        "macOS 10.11 or higher. Your "
                        "program will fail in this "
                        "system."
                    )
            del macos_version
    except ImportError:
        API = os.environ["QT_API"] = "pyqt6"
if API in PYQT6_API:
    try:
        from PyQt6.QtCore import PYQT_VERSION_STR as PYQT_VERSION  #
analysis:ignore
        from PyQt6.QtCore import QT_VERSION_STR as QT_VERSION  # analysis:ignore
        QT5 = PYQT5 = False
        QT6 = PYQT6 = True
    except ImportError:
        API = os.environ["QT_API"] = "pyside2"
if API in PYSIDE2_API:
    try:
        from PySide2 import __version__ as PYSIDE_VERSION  # analysis:ignore
        from PySide2.QtCore import __version__ as QT_VERSION  # analysis:ignore
        PYQT5 = False
        QT5 = PYSIDE2 = True
        if sys.platform == "darwin":
            macos_version = parse(platform.mac_ver()[0])
            if macos_version < parse("10.11"):
                if parse(QT_VERSION) >= parse("5.11"):
                    raise PythonQtError(
                        "Qt 5.11 or higher only works in "
                        "macOS 10.11 or higher. Your "
                        "program will fail in this "
                        "system."
                    )
            del macos_version
    except ImportError:
        API = os.environ["QT_API"] = "pyside6"
if API in PYSIDE6_API:
    try:
        from PySide6 import __version__ as PYSIDE_VERSION  # analysis:ignore
        from PySide6.QtCore import __version__ as QT_VERSION  # analysis:ignore
        QT5 = PYQT5 = False
        QT6 = PYSIDE6 = True
    except ImportError:
        API = os.environ["QT_API"] = "pyqt5"
# If a correct API name is passed to QT_API and it could not be found,
# switches to another and informs through the warning
if API != initial_api and binding_specified:
    warnings.warn(
        'Selected binding "{}" could not be found, '
        'using "{}"'.format(initial_api, API),
        RuntimeWarning,
    )
API_NAME = {
    "pyqt6": "PyQt6",
    "pyqt5": "PyQt5",
    "pyside2": "PySide2",
    "pyside6": "PySide6",
}[API]
try:
    # QtDataVisualization backward compatibility (QtDataVisualization vs.
QtDatavisualization)
```

```
    # Only available for Qt5 bindings > 5.9 on Windows
    from . import QtDataVisualization as QtDatavisualization  # analysis:ignore
except (ImportError, PythonQtError):
    pass
def _warn_old_minor_version(name, old_version, min_version):
    """Warn if using a Qt or binding version no longer supported by QtPy."""
    warning_message = (
        "{name} version {old_version} is not supported by QtPy. "
        "To ensure your application works correctly with QtPy, "
        "please upgrade to {name} {min_version} or later.".format(
            name=name, old_version=old_version, min_version=min_version
        )
    )
    warnings.warn(warning_message, PythonQtWarning)
# Warn if using an End of Life or unsupported Qt API/binding minor version
if QT_VERSION:
    if QT5 and (parse(QT_VERSION) < parse(QT5_VERSION_MIN)):
        _warn_old_minor_version("Qt5", QT_VERSION, QT5_VERSION_MIN)
    elif QT6 and (parse(QT_VERSION) < parse(QT6_VERSION_MIN)):
        _warn_old_minor_version("Qt6", QT_VERSION, QT6_VERSION_MIN)
if PYQT_VERSION:
    if PYQT5 and (parse(PYQT_VERSION) < parse(PYQT5_VERSION_MIN)):
        _warn_old_minor_version("PyQt5", PYQT_VERSION, PYQT5_VERSION_MIN)
    elif PYQT6 and (parse(PYQT_VERSION) < parse(PYQT6_VERSION_MIN)):
        _warn_old_minor_version("PyQt6", PYQT_VERSION, PYQT6_VERSION_MIN)
elif PYSIDE_VERSION:
    if PYSIDE2 and (parse(PYSIDE_VERSION) < parse(PYSIDE2_VERSION_MIN)):
        _warn_old_minor_version("PySide2", PYSIDE_VERSION, PYSIDE2_VERSION_MIN)
    elif PYSIDE6 and (parse(PYSIDE_VERSION) < parse(PYSIDE6_VERSION_MIN)):
        _warn_old_minor_version("PySide6", PYSIDE_VERSION, PYSIDE6_VERSION_MIN)
```

```
#
# Copyright © The Spyder Development Team
#
# Licensed under the terms of the MIT License
# (see LICENSE.txt for details)
import warnings
def introduce_renamed_methods_qheaderview(QHeaderView):
    _isClickable = QHeaderView.isClickable
    def sectionsClickable(self):
        """
        QHeaderView.sectionsClickable() -> bool
        """
        return _isClickable(self)
    QHeaderView.sectionsClickable = sectionsClickable
    def isClickable(self):
        warnings.warn(
            "isClickable is only available in Qt4. Use " "sectionsClickable
instead.",
            stacklevel=2,
        )
        return _isClickable(self)
    QHeaderView.isClickable = isClickable
    _isMovable = QHeaderView.isMovable
    def sectionsMovable(self):
        """
        QHeaderView.sectionsMovable() -> bool
        """
        return _isMovable(self)
    QHeaderView.sectionsMovable = sectionsMovable
    def isMovable(self):
        warnings.warn(
            "isMovable is only available in Qt4. Use " "sectionsMovable
instead.",
            stacklevel=2,
        )
        return _isMovable(self)
    QHeaderView.isMovable = isMovable
    _resizeMode = QHeaderView.resizeMode
    def sectionResizeMode(self, logicalIndex):
        """
        QHeaderView.sectionResizeMode(int) -> QHeaderView.ResizeMode
        """
        return _resizeMode(self, logicalIndex)
    QHeaderView.sectionResizeMode = sectionResizeMode
    def resizeMode(self, logicalIndex):
        warnings.warn(
            "resizeMode is only available in Qt4. Use " "sectionResizeMode 
instead.",
            stacklevel=2,
        )
        return _resizeMode(self, logicalIndex)
    QHeaderView.resizeMode = resizeMode
    _setClickable = QHeaderView.setClickable
    def setSectionsClickable(self, clickable):
        """
        QHeaderView.setSectionsClickable(bool)
        """
        return _setClickable(self, clickable)
    QHeaderView.setSectionsClickable = setSectionsClickable
    def setClickable(self, clickable):
        warnings.warn(
            "setClickable is only available in Qt4. Use "
            "setSectionsClickable instead.",
            stacklevel=2,
        )
        return _setClickable(self, clickable)
    QHeaderView.setClickable = setClickable
    _setMovable = QHeaderView.setMovable
    def setSectionsMovable(self, movable):
        """
        QHeaderView.setSectionsMovable(bool)
        """
        return _setMovable(self, movable)
    QHeaderView.setSectionsMovable = setSectionsMovable
    def setMovable(self, movable):
        warnings.warn(
            "setMovable is only available in Qt4. Use " "setSectionsMovable 
instead.",
            stacklevel=2,
        )
        return _setMovable(self, movable)
    QHeaderView.setMovable = setMovable
    _setResizeMode = QHeaderView.setResizeMode
    def setSectionResizeMode(self, *args):
        """
        QHeaderView.setSectionResizeMode(QHeaderView.ResizeMode)
        QHeaderView.setSectionResizeMode(int, QHeaderView.ResizeMode)
        """
        _setResizeMode(self, *args)
    QHeaderView.setSectionResizeMode = setSectionResizeMode
    def setResizeMode(self, *args):
        warnings.warn(
            "setResizeMode is only available in Qt4. Use "
            "setSectionResizeMode instead.",
            stacklevel=2,
        )
        _setResizeMode(self, *args)
    QHeaderView.setResizeMode = setResizeMode
```

```
## Introduction
"""
Welcome to the tutorial for Pyzo! This tutorial should get you
familiarized with Pyzo in just a few minutes. If you feel this tutorial
contains errors or lacks some information, please let us know via
pyzo@googlegroups.com.
Pyzo is a cross-platform Python IDE focused on interactivity and
introspection, which makes it very suitable for scientific computing.
Its practical design is aimed at simplicity and efficiency.
Pyzo consists of two main components, the editor and the shell, and
uses a set of pluggable tools to help the programmer in various ways.
"""
## The editor
"""
The editor (this window) is where your code is located; it is the central
component of Pyzo.
In the editor, each open file is represented as a tab. By right-clicking on
a tab, files can be run, saved, closed, etc.
The right mouse button also enables one to make a file the MAIN FILE of
a project. This file can be recognized by its star symbol and its blue filename,
and it enables running the file more easily (as we will see later in this
tutorial).
For larger projects, the Project manager tool can be used to manage your files
(also described later in this tutorial)
"""
## The shells
"""
The other main component is the window that holds the shells. When Pyzo
starts, a default shell is created. You can add more shells that run
simultaneously, and which may be of different Python versions.
It is good to know that the shells run in a sub-process, such that
when it is busy, Pyzo itself stays responsive, which allows you to
keep coding and even run code in another shell.
Another notable feature is that Pyzo can integrate the event loop of
five different GUI toolkits, thus enabling interactive plotting with
e.g., Visvis or Matplotlib.
Via "Shell > Edit shell configurations", you can edit and add shell
configurations. This allows you to for example select the initial
directory, or use a custom PYTHONPATH.
"""
## The tools
"""
Via the "Tools" menu, one can select which tools to use. The tools can
```

```
be positioned in any way you want, and can also be un-docked.
Try the "Source Structure" tool to see the outline of this tutorial!
Note that the tools system is designed such that it's quite easy to
create your own tools. Look at the online wiki for more information,
or use one of the existing tools as an example. Also, Pyzo does not
need to restart to see new tools, or to update existing tools.
"""
## Running code
"""
Pyzo supports several ways to run source code in the editor. (see
also the "Run" menu).
  * Run selected lines. If a line is partially selected, the whole
    line is executed. If there is no selection, Pyzo will run the
    current line.
  * Run cell. A cell is everything between two commands starting
    with '##', such as the headings in this tutorial. Try running
    the code at the bottom of this cell!
  * Run file. This runs all the code in the current file.
  * Run project main file. Runs the code in the current project's
    main file.
Additionally, you can run the current file or the current project's
main file as a script. This will first restart the shell to provide
a clean environment. The shell is also initialized differently:
Things done on shell startup in INTERACTIVE MODE:
  * sys.argv = ['']
  * sys.path is prepended with an empty string (current working directory)
  * The working dir is set to the "Initial directory" of the shell config.
  * The PYTHONSTARTUP script is run.
Things done on shell startup in SCRIPT MODE:
  * __file__ = <script_filename>
  * sys.argv = [ <script_filename> ]
  * sys.path is prepended with the directory containing the script.
  * The working dir is set to the directory containing the script.
Depending on the settings of the Project mananger, the current project
directory may also be inserted in sys.path.
"""
a = 3
b = 4
print("The answer is " + str(a + b))
## The menu
"""
Almost all functionality of Pyzo can be accessed via the menu. For more
advanced/specific stuff, you can use the logger tool (see also
Settings > Advanced settings)
All actions in the menu can be accessed via a shortcut. Change the
shortcuts using the shortcut editor: Settings > Edit key mappings.
"""
## Introspection
"""
Pyzo has strong introspection capabilities. Pyzo knows about the objects
in the shell, and parses (not runs) the source code in order to detect
the structure of your code. This enables powerful instospection such
as autocompletion, calltips, interactive help and source structure.
"""
## Debugging
"""
Pyzo supports post-mortem debugging, which means that after something
went wrong, you can inspect the stack trace to find the error.
The easiest way to start debugging is to press the "Debug" button
at the upper right corner of the shells.
Once in debug mode, the button becomes expandable, allowing you to
see the stack trace and go to any frame you like. (Starting debug mode
brings you to the bottom frame.) Changing a frame will make all objects
in that frame available in the shell. If possible, Pyzo will also show
the source file belonging to that frame, and select the line where the
error occurred.
Debugging can also be controlled via magic commands, enter "?" in the
shell for more information.
Below follows an example that you can run to test the debugging.
"""
import random
someModuleVariable = True
def getNumber():
    return random.choice(range(10))
def foo():
    spam = "yum"
    eggs = 7
    value = bar()
def bar():
    total = 0
    for i1 in range(100):
        i2 = getNumber()
        total += i1 / i2
    return total
foo()
## The Project manager
```

```
"""
For working on projects, the Project manager tool can help you to keep
an overview of all your files. To open up the Project manager tool,
select it from the menu (Tools > Project manager).
To add, remove, or edit your projects, click the button with the
wrench icon. In the dialog, select 'New project' to add a project and
select the directory where your project is located. When the project
is added, you can change the Project description (name).
You can select wether the project path is added to the Python
sys.path. This feature allows you to import project modules from the
shell, or from scripts which are not in the project root directory.
Note that this feature requires a restart of the shell to take effect
(Shell > Restart)
The Project manager allows you to switch between your projects easily
using the selection box at the top. The tree view shows the files and
directories in your project. Files can be hidden using the filter that
is specified at the bottom of the Project manager, e.g. !*.pyc to hide
all files that have the extension pyc.
"""
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
"""
This file provides the pyzo history viewer. It contains two main components: the
History class, which is a Qt model, and the PyzoHistoryViewer, which is a Qt
view
"""
import pyzo
from pyzo.qt import QtCore, QtGui, QtWidgets  # noqa
from pyzo import translate
from pyzo.core.menu import Menu
tool_name = translate("pyzoHistoryViewer", "History viewer")
tool_summary = "Shows the last used commands."
class PyzoHistoryViewer(QtWidgets.QWidget):
    """
    The history viewer has several ways of using the data stored in the history:
     - double click a single item to execute in the current shell
     - drag and drop one or multiple selected lines into the editor or any
       other widget or application accepting plain text
     - copy selected items using the copy item in the pyzo edit menu
     - copy selected items using the context menu
     - execute selected items in the current shell using the context menu
    """
    def __init__(self, parent=None):
        super().__init__(parent)
        # Widgets
        self._search = QtWidgets.QLineEdit(self)
        self._list = QtWidgets.QListWidget(self)
        # Set monospace
        font = self._list.font()
        font.setFamily(pyzo.config.view.fontname)
        self._list.setFont(font)
        # Layout
        layout = QtWidgets.QVBoxLayout(self)
        self.setLayout(layout)
        layout.addWidget(self._search, 0)
        layout.addWidget(self._list, 1)
        # set margins
        margin = pyzo.config.view.widgetMargin
        layout.setContentsMargins(margin, margin, margin, margin)
        # Customize line edit
        self._search.setPlaceholderText(translate("menu", "Search"))
        self._search.textChanged.connect(self._on_search)
        # Drag/drop
        self._list.setSelectionMode(self._list.ExtendedSelection)
        self._list.setDragEnabled(True)
        self._list.doubleClicked.connect(self._onDoubleClicked)
        # Context menu
        self._menu = Menu(self, translate("menu", "History"))
        self._menu.addItem(
            translate("menu", "Copy ::: Copy selected lines"),
            pyzo.icons.page_white_copy,
            self.copy,
            "copy",
        )
        self._menu.addItem(
            translate("menu", "Run ::: Run selected lines in current shell"),
            pyzo.icons.run_lines,
            self.runSelection,
            "run",
        )
        self._menu.addItem(
            translate("menu", "Remove ::: Remove selected history items(s)"),
            pyzo.icons.delete,
            self.removeSelection,
            "remove",
        )
        self._list.setContextMenuPolicy(QtCore.Qt.CustomContextMenu)
        self._list.customContextMenuRequested.connect(
            self._onCustomContextMenuRequested
        )
        # Populate
        for command in pyzo.command_history.get_commands():
            self._list.addItem(command)
        # Scroll to end of list on start up
        self._list.setCurrentRow(self._list.count() - 1)
        item = self._list.currentItem()
        self._list.scrollToItem(item)
        # Keep up to date ...
        pyzo.command_history.command_added.connect(self._on_command_added)
        pyzo.command_history.command_removed.connect(self._on_command_removed)
        pyzo.command_history.commands_reset.connect(self._on_commands_reset)
    def _on_search(self):
        needle = self._search.text()
        for i in range(self._list.count()):
            item = self._list.item(i)
            item.setHidden(bool(needle and needle not in item.text()))
    ## Keep track of history
    def _on_command_added(self, command):
        item = QtWidgets.QListWidgetItem(command, self._list)
        self._list.addItem(item)
        needle = self._search.text()
        item.setHidden(bool(needle and needle not in command))
        self._list.scrollToItem(item)
    def _on_command_removed(self, index):
        self._list.takeItem(index)
    def _on_commands_reset(self):
        self._list.clear()
        for command in pyzo.command_history.get_commands():
            self._list.addItem(command)
    ## User actions
    def _onCustomContextMenuRequested(self, pos):
        self._menu.popup(self._list.viewport().mapToGlobal(pos))
    def copy(self, event=None):
        text = "\n".join(i.text() for i in self._list.selectedItems())
        QtWidgets.qApp.clipboard().setText(text)
    def removeSelection(self, event=None):
        indices = [i.row() for i in self._list.selectedIndexes()]
        for i in reversed(sorted(indices)):
            pyzo.command_history.pop(i)
    def runSelection(self, event=None):
        commands = [i.text() for i in self._list.selectedItems()]
        shell = pyzo.shells.getCurrentShell()
        if shell is not None:
            for command in reversed(commands):
                pyzo.command_history.append(command)
            shell.executeCommand("\n".join(commands) + "\n")
            if len(commands) > 1 and commands[-1].startswith(" "):
                shell.executeCommand("\n")  # finalize multi-command
    def _onDoubleClicked(self, index):
        text = "\n".join(i.text() for i in self._list.selectedItems())
        shell = pyzo.shells.getCurrentShell()
        if shell is not None:
            shell.executeCommand(text + "\n")
            # Do not update history? Was this intended?
if __name__ == "__main__":
    import pyzo.core.main
    m = pyzo.core.main.MainWindow()
    view = PyzoHistoryViewer()
    view.show()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
import sys, re
from functools import partial
from pyzo.qt import QtCore, QtGui, QtWidgets  # noqa
import pyzo
tool_name = pyzo.translate("pyzoInteractiveHelp", "Interactive help")
tool_summary = pyzo.translate(
    "pyzoInteractiveHelp", "Shows help on an object when using up/down in
autocomplete."
)
keywordsHelp = {
    "await": pyzo.translate(
        "pyzoInteractiveHelp",
        "Suspend the execution of coroutine on an awaitable object. Can only be
used inside a coroutine function.",
    ),
    "else": pyzo.translate(
        "pyzoInteractiveHelp",
        """This keyword can appear as part of an alternative (see: ``if``), a
loop (see: ``for``, ``while``) or a ``try`` statement.""",
    ),
    "import": pyzo.translate(
        "pyzoInteractiveHelp",
        """Find and load modules or members of modules.
Examples:
```
import foo   # foo imported and bound locally
import foo.bar.baz   # foo.bar.baz imported, foo bound locally
import foo.bar.baz as fbb # foo.bar.baz imported and bound as fbb
from foo.bar import baz   # foo.bar.baz imported and bound as baz
from foo import attr   # foo imported and foo.attr bound as attr
from foo import *   # foo imported and all its members bound
                                    # under their respective names
```""",
    ),
    "pass": pyzo.translate(
        "pyzoInteractiveHelp",
        "This is an instruction that does nothing. It is useful in cases where a
statement must appear because of syntactic constraints but we have nothing to
```

```
do.",
    ),
    "break": pyzo.translate(
        "pyzoInteractiveHelp",
        """This keyword can only appear in the body of a ``for`` or ``while``
loop. It terminates the loop early, regardless of the loop condition.
See also: ``continue``""",
    ),
    "except": pyzo.translate(
        "pyzoInteractiveHelp",
        "An ``except`` clause can appear as a part of a ``try`` statement.",
    ),
    "in": pyzo.translate(
        "pyzoInteractiveHelp",
        """This keyword usually refers to membership of an object in a
structure. There are actually two different kinds of ``in``.
In a construct of the form ``for identifier in iterable``, ``in`` is a purely
syntactic element that bears no meaning per se. See: ``for``
Outside such constructs, ``in`` is an operator and its precise meaning depends
on the type of the first operand.""",
    ),
    "raise": pyzo.translate(
        "pyzoInteractiveHelp",
        "The ``raise`` statement is used to raise an Exception. Raising an
exception will cause the program to abort, unless the exception is handled by a
surrounding ``try``/``except`` statement or ``with`` statement.",
    ),
    "class": pyzo.translate(
        "pyzoInteractiveHelp", "This keyword introduces the definition of a
class."
    ),
    "finally": pyzo.translate(
        "pyzoInteractiveHelp",
        "An ``finally`` clause can appear as a part of a ``try`` statement.",
    ),
    "is": pyzo.translate(
        "pyzoInteractiveHelp",
        """This operator tests for an object’s identity: ``x is y`` is true if
and only if ``x`` and ``y`` are the same object.
See also: ``id``""",
    ),
    "return": pyzo.translate(
        "pyzoInteractiveHelp",
        "This keyword can only appear in the definition of a function. It is 
usually followed by an expression. It means that the execution of the function 
call terminates here and that the result of the call is the value of the 
following expression. If ``return`` is not followed by an expression, the result 
is ``None``.",
    ),
    "and": pyzo.translate(
        "pyzoInteractiveHelp",
        """This operator computes the boolean conjunction in a lazy manner. More 
precisely, the expression ``x and y`` first evaluates ``x``; if ``x`` is false, 
its value is returned; otherwise, ``y`` is evaluated and the resulting value is 
returned.
See also: ``or``, ``not``, ``True``, ``False``""",
    ),
    "continue": pyzo.translate(
        "pyzoInteractiveHelp",
        """This keyword can only appear in the body of a ``for`` or ``while`` 
loop. It terminates the current run of the body early. The loop may still make 
additional runs of its body if its condition is still true .
See also: ``break``""",
    ),
    "for": pyzo.translate(
        "pyzoInteractiveHelp",
        """The ``for`` statement is used to iterate over the elements of an 
iterable object:
```
for variable in iterable:
    body
else: # optional
    suite
```
The expression ``iterable`` is evaluated once; its value should be an iterable 
object. An iterator is created for the result of that expression. The ``body`` 
is then executed once for each item provided by the iterator, in the order 
returned by the iterator: each item in turn is assigned to the ``variable`` and 
then the ``body`` is executed. When the items are exhausted (which is 
immediately when the ``iterable`` is empty), the ``suite`` in the ``else`` 
clause, if present, is executed, and the loop terminates.
A ``break`` statement executed in the first suite terminates the loop without 
executing the ``else`` clause’s ``suite``. A ``continue`` statement executed in 
the first suite skips the rest of the ``body`` and continues with the next item, 
or with the ``else`` clause if there is no next item.
The for-loop makes assignments to the ``variable``. This overwrites all previous
```

~~~
assignments to that variable including those made in the ``body`` of the for-
loop:
```
for i in range(10):
    print(i)
    i = 5             # this will not affect the for-loop
                      # because i will be overwritten with the next
                      # index in the range
```
The ``for`` keyword can also be part of a generator.
Example:
```
[2*x for x in T]
```
This builds from ``T`` a list where each item was doubled.
""",
    ),
    "lambda": pyzo.translate(
        "pyzoInteractiveHelp",
        """This keyword is used to produce an anonymous function.
Example:
```
lambda x,y,z : (x+y) * z
```""",
    ),
    "try": pyzo.translate(
        "pyzoInteractiveHelp",
        """The ``try`` statement allows to recover when an Exception was raised.
It has two main forms.
```
try:
    critical_section
except SomeExceptionType as e:
    suite   # manage exceptions of type SomeExceptionType
else:       # optional
    everything_ok
```
If ``critical_section`` raises an exception of type ``SomeExceptionType``, the
whole program will not abort but the ``suite`` will be executed, with ``e``
being bound to the actual exception. Several ``except`` clauses may be
specified, each dealing with some kind of exception. If an exception is raised
in ``critical_section`` but is not managed by any ``except`` clause, the whole
program aborts. If there is an ``else`` clause, its body is executed if the
~~~

~~~
control flow leaves the ``critical_section``, no exception was raised, and no
``return``, ``continue`` or ``break`` statement was executed. Exceptions in the
``else`` clause are not handled by the preceding ``except`` clauses.
```
try:
    critical_section
finally:
    suite
```
If ``critical_section`` raises an exception, the ``suite`` will be executed
before aborting the program. The ``suite`` is also executed when no exception is
raised. When a ``return``, ``break`` or ``continue`` statement is executed in
the ``critical_section``, the ``finally`` clause is also executed ‘on the way
out.’ A ``continue`` statement is illegal in the ``finally`` clause.
Both mechanisms can be combined, the ``finally`` clause coming after the last
``except`` clause or after the optional ``else`` clause.""",
    ),
    "as": pyzo.translate(
        "pyzoInteractiveHelp",
        "The ``as`` keyword can appear in an ``import`` statement, in an
``except`` clause of a ``try`` statement or in a ``with`` statement.",
    ),
    "def": pyzo.translate(
        "pyzoInteractiveHelp", "This keyword introduces the definition of a
function."
    ),
    "from": pyzo.translate(
        "pyzoInteractiveHelp",
        "This keyword can appear as a part of an ``import`` or a ``raise``. It
has different meanings; see help on those keywords.",
    ),
    "nonlocal": pyzo.translate(
        "pyzoInteractiveHelp",
        """This keyword can only appear in the definition of a function. It is
followed by identifiers and indicates that those identifier refer variables from
the outer scope, not local variables in the function being defined.
See also: ``global``""",
    ),
    "while": pyzo.translate(
        "pyzoInteractiveHelp",
        """The while statement is used for repeated execution as long as an
expression is true:
```
~~~

~~~
while expression:
    body
else:           # optional
    suite
```
This repeatedly tests the ``expression`` and, if it is true, executes the
``body``; if the ``expression`` is false (which may be the first time it is
tested) the ``suite`` of the ``else`` clause, if present, is executed and the
loop terminates.
A ``break`` statement executed in the ``body`` terminates the loop without
executing the ``else`` clause’s ``suite``. A ``continue`` statement executed in
the ``body`` skips the rest of the ``body`` and goes back to testing the
``expression``.""",
    ),
    "assert": pyzo.translate(
        "pyzoInteractiveHelp",
        "Assert statements are a convenient way to insert debugging assertions
into a program.",
    ),
    "del": pyzo.translate(
        "pyzoInteractiveHelp",
        """Deletion of a name removes the binding of that name from the local or
global namespace. It is also possible to delete an item in a list.""",
    ),
    "global": pyzo.translate(
        "pyzoInteractiveHelp",
        """This keyword is followed by identifiers and indicates that those
identifiers refer variables from the module scope, not local variables.
See also: ``nonlocal``""",
    ),
    "not": pyzo.translate(
        "pyzoInteractiveHelp",
        """The operator ``not`` returns ``True`` if its argument is false,
``False`` otherwise.
See also: ``and``, ``or``, ``True``, ``False``""",
    ),
    "with": pyzo.translate(
        "pyzoInteractiveHelp",
        """The ``with`` statement is used to wrap the execution of a block with
methods defined by a context manager.
Example:
```
with open(filename) as infile:
~~~

~~~
    header = infile.readline()
    # etc.
```
In this example, the context manager will ensure correct closing of the file
upon exiting the ``with`` statement, even if an exception was raised.""",
    ),
    "async": pyzo.translate(
        "pyzoInteractiveHelp",
        "This keyword appears as part of the ``async def``, ``async for`` and
``async with`` constructs and allows for writing coroutines.",
    ),
    "elif": pyzo.translate(
        "pyzoInteractiveHelp",
        "This keyword can only appear as part of an alternative. See: ``if``",
    ),
    "if": pyzo.translate(
        "pyzoInteractiveHelp",
        """This keyword usually introduces an alternative. It is followed by a
conditon and introduces the statement to execute when this condition is true.
The alternative can only have one ``if`` clause, at the beginning. It can also
comprise one or more ``elif`` clauses and at most one ``else`` clause at the
end.
Example:
```
if condition1 :
    doSomething       # when condition1 evaluates to True
elif condition2 :     # optional, may have several such clauses
    doSomeOtherthing  # when condition1 evaluates to False and condition2 to
True
else :                # optional
    doAnotherThing    # when all conditions evaluate to False
```
The ``if`` keyword can also appear in a generator.
Example:
```
[x for x in T if x%2==0]
```
This filters ``T`` to keep only its even items.""",
    ),
    "or": pyzo.translate(
        "pyzoInteractiveHelp",
        """This operator computes the boolean disjunction in a lazy manner. More
precisely, the expression ``x or y`` first evaluates ``x``; if ``x`` is true,
~~~

```
its value is returned; otherwise, ``y`` is evaluated and the resulting value is
returned.
See also: ``and``, ``not``, ``True``, ``False``""",
    ),
    "yield": pyzo.translate(
        "pyzoInteractiveHelp",
        """The ``yield`` expression is used when defining a generator function
or an asynchronous generator function and thus can only be used in the body of a
function definition. Using a yield expression in a function’s body causes that
function to be a generator, and using it in an ``async def`` function’s body
causes that coroutine function to be an asynchronous generator.""",
    ),
}
operators = [
    "+",
    "-",
    "*",
    "**",
    "/",
    "//",
    "%",
    "@",
    "<<",
    ">>",
    "&",
    "|",
    "^",
    "~",
    "<",
    ">",
    "<=",
    ">=",
    "==",
    "!=",
]
operatorsHelp = pyzo.translate(
    "pyzoInteractiveHelp",
    "No help is available for operators because they are ambiguous: their
meaning depend on the type of the first operand.",
)
#
htmlWrap = """<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
"http://www.w3.org/TR/REC-html40/strict.dtd">
```

```
<html>
<head>
<style type="text/css">
pre, code {{background-color: #F2F2F2;}}
</style>
</head>
<body style=" font-family:'Sans Serif'; font-size:{}pt; font-weight:400; font-
style:normal;">
{}
</body>
</html>
"""
# Define title text (font-size percentage does not seem to work sadly.)
def get_title_text(objectName, h_class="", h_repr=""):
    title_text = "<p style='background-color:#def;'>"
    if h_class == "~python_keyword~":
        title_text += "<b>Keyword:</b> {}".format(objectName)
    elif h_class == "~python_operator~":
        title_text += "<b>Operator:</b> {}".format(objectName)
    elif h_class == "":
        title_text += "<b>Unknown construct:</b> {}".format(objectName)
    else:
        title_text += "<b>Object:</b> {}".format(objectName)
        title_text += ", <b>class:</b> {}".format(h_class)
        if h_repr:
            if len(h_repr) > 40:
                h_repr = h_repr[:37] + "..."
            title_text += ", <b>repr:</b> {}".format(h_repr)
    # Finish
    title_text += "</p>\n"
    return title_text
initText = pyzo.translate(
    "pyzoInteractiveHelp",
    """
Help information is queried from the current shell
when moving up/down in the autocompletion list
and when double clicking on a name.
""",
)
class PyzoInteractiveHelpHistoryMenu(QtWidgets.QMenu):
    def __init__(self, title, parent, forward):
        super().__init__(title, parent)
        self._forward = forward
        self.aboutToShow.connect(self.populate)
    def populate(self):
        self.clear()
        if self._forward:
            indices = range(self.parent()._histindex + 1,
len(self.parent()._history))
        else:
            indices = range(self.parent()._histindex - 1, -1, -1)
        for i in indices:
            action = self.addAction(self.parent()._history[i])
            action.triggered.connect(partial(self.doAction, i=i))
    def doAction(self, i):
        self.parent()._histindex = i
        self.parent().setObjectName(self.parent().currentHist())
class PyzoInteractiveHelp(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        # Create text field, checkbox, and button
        self._text = QtWidgets.QLineEdit(self)
        self._printBut = QtWidgets.QPushButton("Print", self)
        style = QtWidgets.qApp.style()
        self._backBut = QtWidgets.QToolButton(self)
        self._backBut.setIcon(style.standardIcon(style.SP_ArrowLeft))
        self._backBut.setIconSize(QtCore.QSize(16, 16))
        self._backBut.setPopupMode(self._backBut.DelayedPopup)
        self._backBut.setMenu(
            PyzoInteractiveHelpHistoryMenu("Backward menu", self, False)
        )
        self._forwBut = QtWidgets.QToolButton(self)
        self._forwBut.setIcon(style.standardIcon(style.SP_ArrowRight))
        self._forwBut.setIconSize(QtCore.QSize(16, 16))
        self._forwBut.setPopupMode(self._forwBut.DelayedPopup)
        self._forwBut.setMenu(
            PyzoInteractiveHelpHistoryMenu("Forward menu", self, True)
        )
        # Create options button
        self._options = QtWidgets.QToolButton(self)
        self._options.setIcon(pyzo.icons.wrench)
        self._options.setIconSize(QtCore.QSize(16, 16))
        self._options.setPopupMode(self._options.InstantPopup)
        self._options.setToolButtonStyle(QtCore.Qt.ToolButtonTextBesideIcon)
        # Create options menu
        self._options._menu = QtWidgets.QMenu()
        self._options.setMenu(self._options._menu)
        # Create browser
        self._browser = QtWidgets.QTextBrowser(self)
        self._browser_text = initText
        self._browser.setContextMenuPolicy(QtCore.Qt.CustomContextMenu)
        self._browser.customContextMenuRequested.connect(self.showMenu)
        # Create two sizers
        self._sizer1 = QtWidgets.QVBoxLayout(self)
        self._sizer2 = QtWidgets.QHBoxLayout()
        # Put the elements together
        self._sizer2.addWidget(self._backBut, 1)
        self._sizer2.addWidget(self._forwBut, 2)
        self._sizer2.addWidget(self._text, 4)
        self._sizer2.addWidget(self._printBut, 0)
        self._sizer2.addWidget(self._options, 3)
        #
        self._sizer1.addLayout(self._sizer2, 0)
        self._sizer1.addWidget(self._browser, 1)
        #
        self._sizer1.setSpacing(2)
        # set margins
        margin = pyzo.config.view.widgetMargin
        self._sizer1.setContentsMargins(margin, margin, margin, margin)
        self.setLayout(self._sizer1)
        # Set config
        toolId = self.__class__.__name__.lower()
        self._config = config = pyzo.config.tools[toolId]
        #
        if not hasattr(config, "smartNewlines"):
            config.smartNewlines = True
        if not hasattr(config, "fontSize"):
            if sys.platform == "darwin":
                config.fontSize = 12
            else:
                config.fontSize = 10
        # Create callbacks
        self._text.returnPressed.connect(self.queryDoc)
        self._printBut.clicked.connect(self.printDoc)
        self._backBut.clicked.connect(self.goBack)
        self._forwBut.clicked.connect(self.goForward)
        #
        self._options.pressed.connect(self.onOptionsPress)
        self._options._menu.triggered.connect(self.onOptionMenuTiggered)
        # Start
        self._history = []
        self._histindex = 0
        self.setText()  # Set default text
        self.onOptionsPress()  # Fill menu
    def showMenu(self, pos):
        menu = self._browser.createStandardContextMenu()
        help = QtWidgets.QAction(
            pyzo.icons.help, pyzo.translate("pyzoInteractiveHelp", "Help on 
this"), menu
        )
        help.triggered.connect(partial(self.helpOnThis, pos=pos))
        menu.insertAction(menu.actions()[0], help)
        menu.exec(self.mapToGlobal(pos))
    def helpOnThis(self, pos):
        name = self._browser.textCursor().selectedText().strip()
        if name == "":
            cursor = self._browser.cursorForPosition(pos)
            cursor.select(cursor.WordUnderCursor)
            name = cursor.selectedText()
        if name != "":
            self.setObjectName(name, True)
    def onOptionsPress(self):
        """Create the menu for the button, Do each time to make sure
        the checks are right."""
        # Get menu
        menu = self._options._menu
        menu.clear()
        # Add smart format option
        action = menu.addAction(pyzo.translate("pyzoInteractiveHelp", "Smart 
format"))
        action._what = "smart"
        action.setCheckable(True)
        action.setChecked(bool(self._config.smartNewlines))
        # Add delimiter
        menu.addSeparator()
        # Add font size options
        currentSize = self._config.fontSize
        for i in range(8, 15):
            action = menu.addAction(
                pyzo.translate("pyzoInteractiveHelp", "Font size: %i") % i
            )
            action._what = "font-size: %ipx" % i
            action.setCheckable(True)
            action.setChecked(i == currentSize)
    def onOptionMenuTiggered(self, action):
        """The user decides what to show in the structure."""
        # Get text
        text = action._what.lower()
        if "smart" in text:
            # Swap value
            current = bool(self._config.smartNewlines)
            self._config.smartNewlines = not current
            # Update
            self.queryDoc()
        elif "size" in text:
            # Get font size
            size = int(text.split(":", 1)[1][:-2])
            # Update
            self._config.fontSize = size
            # Update
            self.setText()
    def setText(self, text=None):
        # (Re)store text
        if text is None:
            text = self._browser_text
        else:
            self._browser_text = text
        # Set text with html header
        size = self._config.fontSize
        self._browser.setHtml(htmlWrap.format(size, text))
    def setObjectName(self, name, addToHist=False):
        """Set the object name programatically
        and query documentation for it."""
        self._text.setText(name)
        self.queryDoc(addToHist)
    def helpFromCompletion(self, name, addToHist=False):
        self.setObjectName(name, addToHist)
    def currentHist(self):
        try:
            return self._history[self._histindex]
        except Exception:
            return None
    def addToHist(self, name):
        if name == self.currentHist():
            return
        self._history = self._history[: self._histindex + 1]
        self._history.append(name)
        self._histindex = len(self._history) - 1
    def restoreCurrent(self):
        self.setObjectName(self.currentHist())
    def goBack(self):
        if self._histindex > 0 and self._history != []:
            self._histindex -= 1
            self.setObjectName(self.currentHist())
    def goForward(self):
        if self._histindex < len(self._history) - 1:
            self._histindex += 1
            self.setObjectName(self.currentHist())
    def printDoc(self):
        """Print the doc for the text in the line edit."""
        # Get name
        name = self._text.text()
        # Tell shell to print doc
        shell = pyzo.shells.getCurrentShell()
        if shell and name:
            if name in operators:
                shell.processLine(
                    'print("""{}""")'.format(
                        "Help on operator: " + name + "\n\n" + operatorsHelp
                    )
                )
            elif name in keywordsHelp:
                shell.processLine(
                    'print("""{}""")'.format(
                        "Help on keyword: " + name + "\n\n" + keywordsHelp[name]
                    )
                )
            else:
                shell.processLine("print({}.__doc__)".format(name))
    def queryDoc(self, addToHistory=True):
        """Query the doc for the text in the line edit."""
        # Get name
        name = self._text.text()
        if addToHistory:
            self.addToHist(name)
        if name in operators:
            text = name + "\n~python_operator~\n\n\n" + operatorsHelp
            self.displayResponse(text)
        elif name in keywordsHelp:
            text = name + "\n~python_keyword~\n\n\n" + keywordsHelp[name]
            self.displayResponse(text)
        else:
            # Get shell and ask for the documentation
            shell = pyzo.shells.getCurrentShell()
            if shell and name:
                future = shell._request.doc(name)
                future.add_done_callback(self.queryDoc_response)
            elif not name:
                self.setText(initText)
    def queryDoc_response(self, future):
        """Process the response from the shell."""
        # Process future
        if future.cancelled():
            # print('Introspect cancelled') # No living kernel
            return
        elif future.exception():
            print("Introspect-queryDoc-exception: ", future.exception())
            return
        else:
            response = future.result()
            if not response:
                return
            self.displayResponse(response)
    def displayResponse(self, response):
        try:
            # Get parts
            parts = response.split("\n")
            objectName, h_class, h_fun, h_repr = tuple(parts[:4])
            h_text = "\n".join(parts[4:])
            # Obtain newlines that we hid for repr
            h_repr.replace("/r", "/n")
            # Make all newlines \n in h_text and strip
            h_text = h_text.replace("\r\n", "\n").replace("\r", "\n")
            h_text = h_text.lstrip()
            # Init text
            text = ""
            # These signs will fool the html
            h_repr = h_repr.replace("<", "&lt;")
            h_repr = h_repr.replace(">", "&gt;")
            h_text = h_text.replace("<", "&lt;")
            h_text = h_text.replace(">", "&gt;")
            if self._config.smartNewlines:
                # Make sure the signature is separated from the rest using at
                # least two newlines
                header = ""
                if True:
                    # Get short version of objectName
                    name = objectName.split(".")[-1]
                    # Is the signature in the docstring?
                    docs = h_text.replace("\n", "|")
                    tmp = re.search("[a-zA-z_\.]*?" + name + "\(.*?\)", docs)
                    if tmp and tmp.span(0)[0] < 5:
                        header = tmp.group(0)
                        h_text = h_text[len(header) :].lstrip(":").lstrip()
                        header = header.replace("|", "")
                        # h_text = header + '\n\n' + h_text
                    elif h_text.startswith(objectName) or
h_text.startswith(name):
                        header, sep, docs = h_text.partition("\n")
                        # h_text = header + '\n\n' + docs
                        h_text = docs
                    elif h_fun is not None and h_fun != "":
                        header = h_fun
                # Parse the text as rest/numpy like docstring
                h_text = self.smartFormat(h_text)
                h_text = re.sub("``(.*?)``", r"<code>\1</code>", h_text)
                if header:
                    h_text = "<p style='color:#005;'><b>%s</b></p>\n%s" % (
                        header,
                        h_text,
                    )
                    # h_text = "<b>%s</b><br /><br />\n%s" % (header, h_text)
            else:
                # Make newlines html
                h_text = h_text.replace("\n", "<br />")
            # Compile rich text
            text += get_title_text(objectName, h_class, h_repr)
            if not self._config.smartNewlines and h_fun is not None and h_fun !=
"":
                text += "<p><b>Signature:</b> {}</p>".format(h_fun)
            text += "{}<br />".format(h_text)
        except Exception:
            try:
                text += get_title_text(objectName, h_class, h_repr)
                if h_fun is not None and h_fun != "":
                    text += "<p><b>Signature:</b> {}</p>".format(h_fun)
                text += h_text
            except Exception:
                text = response
        # Done
        # size = self._config.fontSize
        self.setText(text)
    def smartFormat(self, text):
        # Get lines
        lines = text.splitlines(True)
        # Test minimal indentation
        minIndent = 9999
        for line in lines[1:]:
            line_ = line.lstrip()
            indent = len(line) - len(line_)
            if line_:
                minIndent = min(minIndent, indent)
        # Remove minimal indentation
        lines2 = [lines[0]]
        for line in lines[1:]:
            lines2.append(line[minIndent:])
        # Prepare
        prevLine_ = ""
        prevIndent = 0
        prevWasHeader = False
        inExample = False
        forceNewline = False
        inPre = False
        # Format line by line
        lines3 = []
        for line in lines2:
            # Get indentation
            line_ = line.lstrip()
            if line_ in ("```", "```\n"):
                if inPre:
                    line = "</pre>"
                    inPre = False
                else:
                    line = "<pre>"
                    inPre = True
            elif not inPre:
                indent = len(line) - len(line_)
```

```
                # indentPart = line[:indent-minIndent]
                indentPart = line[:indent]
                if not line_:
                    lines3.append("<br />")
                    forceNewline = True
                    continue
                # Indent in html
                line = "&nbsp;" * len(indentPart) + line
                # Determine if we should introduce a newline
                isHeader = False
                if ("---" in line or "===" in line) and indent == prevIndent:
                    # Header
                    lines3[-1] = "<b>" + lines3[-1] + "</b>"
                    line = ""  #'<br /> ' + line
                    isHeader = True
                    inExample = False
                    # Special case, examples
                    if prevLine_.lower().startswith("example"):
                        inExample = True
                    else:
                        inExample = False
                elif " : " in line:
                    tmp = line.split(" : ", 1)
                    line = "<br /><u>" + tmp[0] + "</u> : " + tmp[1]
                elif line_.startswith("* "):
                    line = "<br />&nbsp;&nbsp;&nbsp;&#8226;" + line_[2:]
                elif prevWasHeader or inExample or forceNewline:
                    line = "<br />" + line
                else:
                    if prevLine_:
                        line = " " + line_
                    else:
                        line = line_
                # Force next line to be on a new line if using a colon
                if " : " in line:
                    forceNewline = True
                else:
                    forceNewline = False
                # Prepare for next line
                prevLine_ = line_
                prevIndent = indent
                prevWasHeader = isHeader
            # Done with line
            lines3.append(line)
        # Done formatting
        return "".join(lines3)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
import sys, os, code
import pyzo
from pyzo.qt import QtCore, QtGui, QtWidgets  # noqa
from pyzo.core.shell import BaseShell
from pyzo.core.pyzoLogging import splitConsole
tool_name = pyzo.translate("pyzoLogger", "Logger")
tool_summary = "Logs messages, warnings and errors within Pyzo."
class PyzoLogger(QtWidgets.QWidget):
    """PyzoLogger
    The main widget for this tool.
    """
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        # logger widget
        self._logger_shell = PyzoLoggerShell(self)
        # set layout
        self.layout = QtWidgets.QVBoxLayout(self)
        self.layout.addWidget(self._logger_shell, 1)
        # spacing of widgets
        self.layout.setSpacing(0)
        # set margins
        margin = pyzo.config.view.widgetMargin
        self.layout.setContentsMargins(margin, margin, margin, margin)
        self.setLayout(self.layout)
    def updateZoom(self):
        self._logger_shell.setZoom(pyzo.config.view.zoom)
    def updateFont(self):
        self._logger_shell.setFont(pyzo.config.view.fontname)
class PyzoLoggerShell(BaseShell):
    """Shell that logs all messages produced by pyzo. It also
    allows to look inside pyzo, which can be handy for debugging
    and developing.
    """
    def __init__(self, parent):
        BaseShell.__init__(self, parent)
        # Set style to Python, or autocompletion does not work
        self.setParser("python")
        # Change background color to make the logger look different from shell
        # Use color as if all lines are highlighted
        f1 = self.getStyleElementFormat("Editor.text")
        f2 = self.getStyleElementFormat("Editor.Highlight current line")
        newStyle = "back:%s, fore:%s" % (f2.back.name(), f1.fore.name())
        self.setStyle(editor_text=newStyle)
        # Create namespace for logger interpreter
        locals = {"pyzo": pyzo, "sys": sys, "os": os}
        # Include linguist tools
        for name in ["linguist", "lrelease", "lupdate", "lhelp"]:
            locals[name] = getattr(pyzo.util._locale, name)
        # Create interpreter to run code
        self._interpreter = code.InteractiveConsole(locals, "<logger>")
        # Show welcome text
        moreBanner = "This is the Pyzo logger shell."
        self.write(
            "Python %s on %s - %s\n\n" % (sys.version[:5], sys.platform,
moreBanner)
        )
        self.write(str(sys.ps1), 2)
        # Split console
        history = splitConsole(self.write, self.writeErr)
        self.write(history)
    def executeCommand(self, command):
        """Execute the command here!"""
        # Use writeErr rather than sys.stdout.write. This prevents
        # the prompts to be logged by the history. Because if they
        # are, the text does not look good due to missing newlines
        # when loading the history.
        # "Echo" stdin
        self.write(command, 1)
        more = self._interpreter.push(command.rstrip("\n"))
        if more:
            self.write(str(sys.ps2), 2)
        else:
            self.write(str(sys.ps1), 2)
    def writeErr(self, msg):
        """This is what the logger uses to write errors."""
        self.write(msg, 0, "#C00")
    # Note that I did not (yet) implement calltips
    def processAutoComp(self, aco):
        """Processes an autocomp request using an AutoCompObject instance."""
        # Try using buffer first
        if aco.tryUsingBuffer():
            return
        # Include buildins?
        if not aco.name:
            command = "__builtins__.keys()"
            try:
                names = eval(command, {}, self._interpreter.locals)
                aco.addNames(names)
            except Exception:
                pass
        # Query list of names
        command = "dir({})".format(aco.name)
        try:
            names = eval(command, {}, self._interpreter.locals)
            aco.addNames(names)
        except Exception:
            pass
        # Done
        aco.finish()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
import pyzo
from pyzo.qt import QtCore, QtGui, QtWidgets
from pyzo import translate
tool_name = translate("pyzoSourceStructure", "Source structure")
tool_summary = "Shows the structure of your source code."
class Navigation:
    def __init__(self):
        self.back = []
        self.forward = []
class PyzoSourceStructure(QtWidgets.QWidget):
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        # Make sure there is a configuration entry for this tool
        # The pyzo tool manager makes sure that there is an entry in
        # config.tools before the tool is instantiated.
        toolId = self.__class__.__name__.lower()
        self._config = pyzo.config.tools[toolId]
        if not hasattr(self._config, "showTypes"):
            self._config.showTypes = ["class", "def", "cell", "todo"]
        if not hasattr(self._config, "level"):
            self._config.level = 2
        # Keep track of clicks so we can "go back"
        self._nav = {}  # editor-id -> Navigation object
        # Create buttons for navigation
        self._navbut_back = QtWidgets.QToolButton(self)
        self._navbut_back.setIcon(pyzo.icons.arrow_left)
        self._navbut_back.setIconSize(QtCore.QSize(16, 16))
        self._navbut_back.setStyleSheet("QToolButton { border: none; padding: 
0px; }")
        self._navbut_back.clicked.connect(self.onNavBack)
        #
        self._navbut_forward = QtWidgets.QToolButton(self)
        self._navbut_forward.setIcon(pyzo.icons.arrow_right)
        self._navbut_forward.setIconSize(QtCore.QSize(16, 16))
        self._navbut_forward.setStyleSheet(
            "QToolButton { border: none; padding: 0px; }"
        )
        self._navbut_forward.clicked.connect(self.onNavForward)
        # # Create icon for slider
        # self._sliderIcon = QtWidgets.QToolButton(self)
        # self._sliderIcon.setIcon(pyzo.icons.text_align_right)
        # self._sliderIcon.setIconSize(QtCore.QSize(16,16))
        # self._sliderIcon.setStyleSheet("QToolButton { border: none; padding: 
0px; }")
        # Create slider
        self._slider = QtWidgets.QSlider(QtCore.Qt.Horizontal, self)
        self._slider.setTickPosition(QtWidgets.QSlider.TicksBelow)
        self._slider.setSingleStep(1)
        self._slider.setPageStep(1)
        self._slider.setRange(1, 5)
        self._slider.setValue(self._config.level)
        self._slider.valueChanged.connect(self.updateStructure)
        # Create options button
        # self._options = QtWidgets.QPushButton(self)
        # self._options.setText('Options'))
        # self._options.setToolTip("What elements to show.")
        self._options = QtWidgets.QToolButton(self)
        self._options.setIcon(pyzo.icons.filter)
        self._options.setIconSize(QtCore.QSize(16, 16))
        self._options.setPopupMode(self._options.InstantPopup)
        self._options.setToolButtonStyle(QtCore.Qt.ToolButtonTextBesideIcon)
        # Create options menu
        self._options._menu = QtWidgets.QMenu()
        self._options.setMenu(self._options._menu)
        # Create tree widget
        self._tree = QtWidgets.QTreeWidget(self)
        self._tree.setHeaderHidden(True)
        self._tree.itemCollapsed.connect(self.updateStructure)  # keep expanded
        self._tree.itemClicked.connect(self.onItemClick)
        # Create two sizers
        self._sizer1 = QtWidgets.QVBoxLayout(self)
        self._sizer2 = QtWidgets.QHBoxLayout()
        self._sizer1.setSpacing(2)
        # set margins
        margin = pyzo.config.view.widgetMargin
        self._sizer1.setContentsMargins(margin, margin, margin, margin)
        # Set layout
        self._sizer1.addLayout(self._sizer2, 0)
        self._sizer1.addWidget(self._tree, 1)
        # self._sizer2.addWidget(self._sliderIcon, 0)
        self._sizer2.addWidget(self._navbut_back, 0)
        self._sizer2.addWidget(self._navbut_forward, 0)
        self._sizer2.addStretch(1)
        self._sizer2.addWidget(self._slider, 6)
        self._sizer2.addStretch(1)
        self._sizer2.addWidget(self._options, 0)
        #
        self.setLayout(self._sizer1)
        # Init current-file name
        self._currentEditorId = 0
        # Bind to events
        pyzo.editors.currentChanged.connect(self.onEditorsCurrentChanged)
        pyzo.editors.parserDone.connect(self.updateStructure)
        self._options.pressed.connect(self.onOptionsPress)
        self._options._menu.triggered.connect(self.onOptionMenuTiggered)
        # Start
        # When the tool is loaded, the editorStack is already done loading
        # all previous files and selected the appropriate file.
        self.onOptionsPress()  # Create menu now
        self.onEditorsCurrentChanged()
    def onOptionsPress(self):
        """Create the menu for the button, Do each time to make sure
        the checks are right."""
        # Get menu
        menu = self._options._menu
        menu.clear()
        for type in ["class", "def", "cell", "todo", "import", "attribute"]:
            checked = type in self._config.showTypes
            action = menu.addAction("Show %s" % type)
            action.setCheckable(True)
            action.setChecked(checked)
    def onOptionMenuTiggered(self, action):
        """The user decides what to show in the structure."""
        # What to show
        type = action.text().split(" ", 1)[1]
        # Swap
        if type in self._config.showTypes:
            while type in self._config.showTypes:
                self._config.showTypes.remove(type)
        else:
            self._config.showTypes.append(type)
        # Update
        self.updateStructure()
    def onEditorsCurrentChanged(self):
        """Notify that the file is being parsed and make
        sure that not the structure of a previously selected
        file is shown."""
        # Get editor and clear list
        editor = pyzo.editors.getCurrentEditor()
        self._tree.clear()
        if editor is None:
            # Set editor id
            self._currentEditorId = 0
        if editor is not None:
            # Set editor id
            self._currentEditorId = id(editor)
            # Notify
            text = translate("pyzoSourceStructure", "Parsing ") + editor._name +
" ..."
            QtWidgets.QTreeWidgetItem(self._tree, [text])
            # Try getting the  structure right now
            self.updateStructure()
    def _getCurrentNav(self):
        if not self._currentEditorId:
            return None
        if self._currentEditorId not in self._nav:
            self._nav[self._currentEditorId] = Navigation()
        return self._nav[self._currentEditorId]
    def onNavBack(self):
        nav = self._getCurrentNav()
        if not nav or not nav.back:
            return
        linenr = nav.back.pop(-1)
        old_linenr = self._navigate_to_line(linenr)
        if old_linenr is not None:
            nav.forward.append(old_linenr)
    def onNavForward(self):
        nav = self._getCurrentNav()
        if not nav or not nav.forward:
            return
        linenr = nav.forward.pop(-1)
        old_linenr = self._navigate_to_line(linenr)
        if old_linenr is not None:
            nav.back.append(old_linenr)
    def onItemClick(self, item):
        """Go to the right line in the editor and give focus."""
        # If item is attribute, get parent
        if not item.linenr:
            item = item.parent()
        old_linenr = self._navigate_to_line(item.linenr)
        if old_linenr is not None:
            nav = self._getCurrentNav()
            if nav and (not nav.back or nav.back[-1] != old_linenr):
                nav.back.append(old_linenr)
                nav.forward = []
    def _navigate_to_line(self, linenr):
        # Get editor
        editor = pyzo.editors.getCurrentEditor()
        if not editor:
            return None
        # Keep current line nr
        old_linenr = editor.textCursor().blockNumber() + 1
        # Move to line
        editor.gotoLine(linenr)
        # Give focus
        pyzo.callLater(editor.setFocus)
        return old_linenr
    def updateStructure(self):
        """Updates the tree."""
        # Get editor
        editor = pyzo.editors.getCurrentEditor()
        if not editor:
            return
        # Something to show
        result = pyzo.parser._getResult()
        if result is None:
            return
        # Do the ids match?
        id0, id1, id2 = self._currentEditorId, id(editor), result.editorId
        if id0 != id1 or id0 != id2:
            return
        # Get current line number and the structure
        ln = editor.textCursor().blockNumber()
        ln += 1  # is ln as in line number area
        def get_color(name, sub="fore"):
            parts = [part.partition(":") for part in theme[name].split(",")]
            colors = {k.strip(): v.strip() for k, _, v in parts}
            return colors[sub]
        try:
            theme = pyzo.themes[pyzo.config.settings.theme.lower()]["data"]
            colours = {
                "cell": get_color("syntax.python.cellcomment"),
                "class": get_color("syntax.classname"),
                "def": get_color("syntax.functionname"),
                "attribute": get_color("syntax.comment"),
                "import": get_color("syntax.keyword"),
                "todo": get_color("syntax.todocomment"),
                "nameismain": get_color("syntax.keyword"),
                "background": get_color("editor.text", "back"),
            }
        except Exception as err:
            print("Reverting to defaut source structure colors:", str(err))
            colours = {
                "cell": "#b58900",
                "class": "#cb4b16",
                "def": "#073642",
                "attribute": "#657b83",
                "import": "#268bd2",
                "todo": "#d33682",
                "nameismain": "#859900",
                "background": "#fff",
            }
        # Define what to show
        showTypes = self._config.showTypes
        # Define to what level to show (now is also a good time to save)
        showLevel = int(self._slider.value())
        self._config.level = showLevel
        showLevel = showLevel if showLevel < 5 else 99
        # Define function to set items
        selectedItem = [None]
        def SetItems(parentItem, fictiveObjects, level):
            level += 1
            for object in fictiveObjects:
                type = object.type
                if type not in showTypes and type != "nameismain":
                    continue
                # Construct text
                if type == "import":
                    text = "→ %s (%s)" % (object.name, object.text)
                elif type == "todo":
                    text = object.name
                elif type == "nameismain":
                    text = object.text
                elif type == "class":
                    text = object.name
                elif type == "def":
                    text = object.name + "()"
                elif type == "attribute":
                    text = "- " + object.name
                elif type in ("cell", "##", "#%%", "# %%"):
                    type = "cell"
                    text = "## " + object.name + " " * 120
                else:
                    text = "%s %s" % (type, object.name)
                # Create item
                thisItem = QtWidgets.QTreeWidgetItem(parentItem, [text])
                color = QtGui.QColor(colours[object.type])
                thisItem.setForeground(0, QtGui.QBrush(color))
                font = thisItem.font(0)
                font.setBold(True)
                if type == "cell":
                    font.setUnderline(True)
                thisItem.setFont(0, font)
                thisItem.linenr = object.linenr
                # Is this the current item?
                if ln and object.linenr <= ln and object.linenr2 > ln:
                    selectedItem[0] = thisItem
                # Any children that we should display?
                if object.children:
                    SetItems(thisItem, object.children, level)
                # Set visibility
                thisItem.setExpanded(bool(level < showLevel))
        # Go
        self._tree.setStyleSheet("background-color: " + colours["background"] +
";")
        self._tree.setUpdatesEnabled(False)
        self._tree.clear()
        SetItems(self._tree, result.rootItem.children, 0)
        self._tree.setUpdatesEnabled(True)
        # Handle selected item
        selectedItem = selectedItem[0]
        if selectedItem:
            selectedItem.setBackground(0, QtGui.QBrush(QtGui.QColor("#CCC")))
            self._tree.scrollToItem(selectedItem)  # ensure visible
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
import urllib.request, urllib.parse
from pyzo.qt import QtCore, QtWidgets
imported_qtwebkit = True
try:
    from pyzo.qt import QtWebKit
except ImportError:
    imported_qtwebkit = False
import pyzo
tool_name = pyzo.translate("pyzoWebBrowser", "Web browser")
tool_summary = "A very simple web browser."
default_bookmarks = [
    "docs.python.org",
    "scipy.org",
    "doc.qt.nokia.com/4.5/",
    "pyzo.org",
]
class WebView(QtWidgets.QTextBrowser):
    """Inherit the webview class to implement zooming using
    the mouse wheel.
    """
    loadStarted = QtCore.Signal()
    loadFinished = QtCore.Signal(bool)
    def __init__(self, parent):
        QtWidgets.QTextBrowser.__init__(self, parent)
        # Current url
        self._url = ""
        self._history = []
        self._history2 = []
        # Connect
        self.anchorClicked.connect(self.load)
    def wheelEvent(self, event):
        # Zooming does not work for this widget
        if QtCore.Qt.ControlModifier & QtWidgets.qApp.keyboardModifiers():
            self.parent().wheelEvent(event)
        else:
            QtWidgets.QTextBrowser.wheelEvent(self, event)
    def url(self):
        return self._url
    def _getUrlParts(self):
        r = urllib.parse.urlparse(self._url)
        base = r.scheme + "://" + r.netloc
        return base, r.path, r.fragment
    #
    #     def loadCss(self, urls=[]):
    #         urls.append('http://docs.python.org/_static/default.css')
    #         urls.append('http://docs.python.org/_static/pygments.css')
    #         text = ''
    #         for url in urls:
    #             tmp = urllib.request.urlopen(url).read().decode('utf-8')
    #             text += '\n' + tmp
    #         self.document().setDefaultStyleSheet(text)
    def back(self):
        # Get url and update forward history
        url = self._history.pop()
        self._history2.append(self._url)
        # Go there
        url = self._load(url)
    def forward(self):
        if not self._history2:
            return
        # Get url and update forward history
        url = self._history2.pop()
        self._history.append(self._url)
        # Go there
        url = self._load(url)
    def load(self, url):
        # Clear forward history
        self._history2 = []
        # Store current url in history
        while self._url in self._history:
            self._history.remove(self._url)
        self._history.append(self._url)
        # Load
        url = self._load(url)
    def _load(self, url):
        """_load(url)
        Convert url and load page, returns new url.
        """
        # Make url a string
        if isinstance(url, QtCore.QUrl):
            url = str(url.toString())
        # Compose relative url to absolute
        if url.startswith("#"):
            base, path, frag = self._getUrlParts()
            url = base + path + url
        elif "//" not in url:
            base, path, frag = self._getUrlParts()
            url = base + "/" + url.lstrip("/")
        # Try loading
        self.loadStarted.emit()
        self._url = url
        try:
            # print('URL:', url)
            text = urllib.request.urlopen(url).read().decode("utf-8")
            self.setHtml(text)
            self.loadFinished.emit(True)
        except Exception as err:
            self.setHtml(str(err))
            self.loadFinished.emit(False)
        # Set
        return url
class PyzoWebBrowser(QtWidgets.QFrame):
    """The main window, containing buttons, address bar and
    browser widget.
    """
    def __init__(self, parent):
        QtWidgets.QFrame.__init__(self, parent)
        # Init config
        toolId = self.__class__.__name__.lower()
        self._config = pyzo.config.tools[toolId]
        if not hasattr(self._config, "zoomFactor"):
            self._config.zoomFactor = 1.0
        if not hasattr(self._config, "bookMarks"):
            self._config.bookMarks = []
        for item in default_bookmarks:
            if item not in self._config.bookMarks:
                self._config.bookMarks.append(item)
        # Get style object (for icons)
        style = QtWidgets.QApplication.style()
        # Create some buttons
        self._back = QtWidgets.QToolButton(self)
        self._back.setIcon(style.standardIcon(style.SP_ArrowBack))
        self._back.setIconSize(QtCore.QSize(16, 16))
        #
        self._forward = QtWidgets.QToolButton(self)
        self._forward.setIcon(style.standardIcon(style.SP_ArrowForward))
        self._forward.setIconSize(QtCore.QSize(16, 16))
        # Create address bar
        # self._address = QtWidgets.QLineEdit(self)
        self._address = QtWidgets.QComboBox(self)
        self._address.setEditable(True)
        self._address.setInsertPolicy(self._address.NoInsert)
        #
        for a in self._config.bookMarks:
            self._address.addItem(a)
        self._address.setEditText("")
        # Create web view
        if imported_qtwebkit:
            self._view = QtWebKit.QWebView(self)
        else:
            self._view = WebView(self)
        #
        #          self._view.setZoomFactor(self._config.zoomFactor)
        #          settings = self._view.settings()
        #          settings.setAttribute(settings.JavascriptEnabled, True)
        #          settings.setAttribute(settings.PluginsEnabled, True)
        # Layout
        self._sizer1 = QtWidgets.QVBoxLayout(self)
        self._sizer2 = QtWidgets.QHBoxLayout()
        #
        self._sizer2.addWidget(self._back, 0)
        self._sizer2.addWidget(self._forward, 0)
        self._sizer2.addWidget(self._address, 1)
        #
        self._sizer1.addLayout(self._sizer2, 0)
        self._sizer1.addWidget(self._view, 1)
        #
        self._sizer1.setSpacing(2)
        # set margins
        margin = pyzo.config.view.widgetMargin
        self._sizer1.setContentsMargins(margin, margin, margin, margin)
        self.setLayout(self._sizer1)
        # Bind signals
        self._back.clicked.connect(self.onBack)
        self._forward.clicked.connect(self.onForward)
        self._address.lineEdit().returnPressed.connect(self.go)
        self._address.activated.connect(self.go)
        self._view.loadFinished.connect(self.onLoadEnd)
        self._view.loadStarted.connect(self.onLoadStart)
        # Start
        self._view.show()
        self.go("http://docs.python.org")
    def parseAddress(self, address):
        if not address.startswith("http"):
            address = "http://" + address
        return address  # QtCore.QUrl(address, QtCore.QUrl.TolerantMode)
    def go(self, address=None):
        if not isinstance(address, str):
            address = self._address.currentText()
        self._view.load(self.parseAddress(address))
    def onLoadStart(self):
        self._address.setEditText("<loading>")
    def onLoadEnd(self, ok):
        if ok:
            # url = self._view.url()
            # address = str(url.toString())
            if imported_qtwebkit:
                address = self._view.url().toString()
            else:
                address = self._view.url()
        else:
            address = "<could not load page>"
        self._address.setEditText(str(address))
    def onBack(self):
        self._view.back()
    def onForward(self):
        self._view.forward()
    def wheelEvent(self, event):
        if QtCore.Qt.ControlModifier & QtWidgets.qApp.keyboardModifiers():
            # Get amount of scrolling
            degrees = event.delta() / 8.0
            steps = degrees / 15.0
            # Set factor
            factor = self._view.zoomFactor() + steps / 10.0
            if factor < 0.25:
                factor = 0.25
            if factor > 4.0:
                factor = 4.0
            # Store and apply
            self._config.zoomFactor = factor
```

```
#             self._view.setZoomFactor(factor)
else:
    QtWidgets.QFrame.wheelEvent(self, event)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
import pyzo
from pyzo.qt import QtCore, QtGui, QtWidgets
tool_name = pyzo.translate("pyzoWorkspace", "Workspace")
tool_summary = pyzo.translate(
    "pyzoWorkspace", "Lists the variables in the current shell's namespace."
)
def splitName(name):
    """splitName(name)
    Split an object name in parts, taking dots and indexing into account.
    """
    name = name.replace("[", ".[")
    parts = name.split(".")
    return [p for p in parts if p]
def joinName(parts):
    """joinName(parts)
    Join the parts of an object name, taking dots and indexing into account.
    """
    name = ".".join(parts)
    return name.replace(".[", "[")
class WorkspaceProxy(QtCore.QObject):
    """WorkspaceProxy
    A proxy class to handle the asynchonous behaviour of getting information
    from the shell. The workspace tool asks for a certain name, and this
    class notifies when new data is available using a qt signal.
    """
    haveNewData = QtCore.Signal()
    def __init__(self):
        QtCore.QObject.__init__(self)
        # Variables
        self._variables = []
        # Element to get more info of
        self._name = ""
        # Bind to events
        pyzo.shells.currentShellChanged.connect(self.onCurrentShellChanged)
pyzo.shells.currentShellStateChanged.connect(self.onCurrentShellStateChanged)
        # Initialize
        self.onCurrentShellStateChanged()
    def addNamePart(self, part):
        """addNamePart(part)
        Add a part to the name.
        """
        parts = splitName(self._name)
        parts.append(part)
        self.setName(joinName(parts))
    def setName(self, name):
        """setName(name)
        Set the name that we want to know more of.
        """
        self._name = name
        shell = pyzo.shells.getCurrentShell()
        if shell:
            future = shell._request.dir2(self._name)
            future.add_done_callback(self.processResponse)
    def goUp(self):
        """goUp()
        Cut the last part off the name.
        """
        parts = splitName(self._name)
        if parts:
            parts.pop()
        self.setName(joinName(parts))
    def onCurrentShellChanged(self):
        """onCurrentShellChanged()
        When no shell is selected now, update this. In all other cases,
        the onCurrentShellStateChange will be fired too.
        """
        shell = pyzo.shells.getCurrentShell()
        if not shell:
            self._variables = []
            self.haveNewData.emit()
    def onCurrentShellStateChanged(self):
        """onCurrentShellStateChanged()
        Do a request for information!
        """
        shell = pyzo.shells.getCurrentShell()
        if not shell:
            # Should never happen I think, but just to be sure
            self._variables = []
        elif shell._state.lower() != "busy":
            future = shell._request.dir2(self._name)
            future.add_done_callback(self.processResponse)
    def processResponse(self, future):
        """processResponse(response)
        We got a response, update our list and notify the tree.
        """
        response = []
        # Process future
        if future.cancelled():
            pass  # print('Introspect cancelled') # No living kernel
        elif future.exception():
            print("Introspect-queryDoc-exception: ", future.exception())
        else:
            response = future.result()
        self._variables = response
        self.haveNewData.emit()
class WorkspaceItem(QtWidgets.QTreeWidgetItem):
    def __lt__(self, otherItem):
        column = self.treeWidget().sortColumn()
        try:
            return float(self.text(column).strip("[]")) > float(
                otherItem.text(column).strip("[]")
            )
        except ValueError:
            return self.text(column) > otherItem.text(column)
class WorkspaceTree(QtWidgets.QTreeWidget):
    """WorkspaceTree
    The tree that displays the items in the current namespace.
    I first thought about implementing this using the mode/view
    framework, but it is so much work and I can't seem to fully
    understand how it works :(
    The QTreeWidget is so very simple and enables sorting very
    easily, so I'll stick with that ...
    """
    def __init__(self, parent):
        QtWidgets.QTreeWidget.__init__(self, parent)
        self._config = parent._config
        # Set header stuff
        self.setHeaderHidden(False)
        self.setColumnCount(3)
        self.setHeaderLabels(
            [
                pyzo.translate("pyzoWorkspace", "Name"),
                pyzo.translate("pyzoWorkspace", "Type"),
                pyzo.translate("pyzoWorkspace", "Repr"),
            ]
        )
        # self.setColumnWidth(0, 100)
        self.setSortingEnabled(True)
        # Nice rows
        self.setAlternatingRowColors(True)
        self.setRootIsDecorated(False)
        # Create proxy
        self._proxy = WorkspaceProxy()
        self._proxy.haveNewData.connect(self.fillWorkspace)
        # For menu
        self.setContextMenuPolicy(QtCore.Qt.DefaultContextMenu)
        self._menu = QtWidgets.QMenu()
        self._menu.triggered.connect(self.contextMenuTriggered)
        # Bind to events
        self.itemActivated.connect(self.onItemExpand)
        self._startUpVariables = ["In", "Out", "exit", "get_ipython", "quit"]
    def contextMenuEvent(self, event):
        """contextMenuEvent(event)
        Show the context menu.
        """
        QtWidgets.QTreeView.contextMenuEvent(self, event)
        # Get if an item is selected
        item = self.currentItem()
        if not item:
            return
        # Create menu
        self._menu.clear()
        commands = [
            ("Show namespace", pyzo.translate("pyzoWorkspace", "Show
namespace")),
            ("Show help", pyzo.translate("pyzoWorkspace", "Show help")),
            ("Delete", pyzo.translate("pyzoWorkspace", "Delete")),
        ]
        for a, display in commands:
            action = self._menu.addAction(display)
            action._what = a
            parts = splitName(self._proxy._name)
            parts.append(item.text(0))
            action._objectName = joinName(parts)
            action._item = item
        # Show
        self._menu.popup(QtGui.QCursor.pos() + QtCore.QPoint(3, 3))
    def contextMenuTriggered(self, action):
        """contextMenuTriggered(action)
        Process a request from the context menu.
        """
        # Get text
        req = action._what.lower()
        if "namespace" in req:
            # Go deeper
            self.onItemExpand(action._item)
        elif "help" in req:
            # Show help in help tool (if loaded)
            hw = pyzo.toolManager.getTool("pyzointeractivehelp")
            if hw:
                hw.setObjectName(action._objectName, addToHist=True)
        elif "delete" in req:
            # Delete the variable
            shell = pyzo.shells.getCurrentShell()
            if shell:
                shell.processLine("del " + action._objectName)
    def onItemExpand(self, item):
        """onItemExpand(item)
        Inspect the attributes of that item.
        """
        self._proxy.addNamePart(item.text(0))
    def fillWorkspace(self):
        """fillWorkspace()
        Update the workspace tree.
        """
        # Clear first
        self.clear()
        # Set name
        line = self.parent()._line
        line.setText(self._proxy._name)
        # Add elements
        for des in self._proxy._variables:
            # Get parts
            parts = list(des)
            if len(parts) < 4:
                continue
            name = parts[0]
            # Pop the 'kind' element
            kind = parts.pop(2)
            # <kludge 2>
            # the typeTranslation dictionary contains "synonyms" for types that 
will be hidden
            # Currently only "method"->"function" is used
            # the try:... is there to have a minimal translation dictionary.
            try:
                kind = self._config.typeTranslation[kind]
            except KeyError:
                pass
            # </kludge 2>
            if kind in self._config.hideTypes:
                continue
            if name.startswith("_") and "private" in self._config.hideTypes:
                continue
            if "startup" in self._config.hideTypes and name in 
self._startUpVariables:
                continue
            # Create item
            item = WorkspaceItem(parts, 0)
            self.addTopLevelItem(item)
            # Set tooltip
            tt = "%s: %s" % (parts[0], parts[-1])
            item.setToolTip(0, tt)
            item.setToolTip(1, tt)
            item.setToolTip(2, tt)
        self.parent().displayEmptyWorkspace(
            self.topLevelItemCount() == 0 and self._proxy._name == ""
        )
class PyzoWorkspace(QtWidgets.QWidget):
    """PyzoWorkspace
    The main widget for this tool.
    """
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        # Make sure there is a configuration entry for this tool
        # The pyzo tool manager makes sure that there is an entry in
        # config.tools before the tool is instantiated.
        toolId = self.__class__.__name__.lower()
        self._config = pyzo.config.tools[toolId]
        if not hasattr(self._config, "hideTypes"):
            self._config.hideTypes = []
        # <kludge 2>
        # configuring the typeTranslation dictionary
        if not hasattr(self._config, "typeTranslation"):
            # to prevent the exception to be raised, one could init to :
            # {"method": "function", "function": "function", "type": "type",
"private": "private", "module": "module"}
            self._config.typeTranslation = {}
        # Defaults
        self._config.typeTranslation["method"] = "function"
        self._config.typeTranslation["builtin_function_or_method"] = "function"
        # <kludge 2>
        # Create tool button
        self._up = QtWidgets.QToolButton(self)
        style = QtWidgets.qApp.style()
        self._up.setIcon(style.standardIcon(style.SP_ArrowLeft))
        self._up.setIconSize(QtCore.QSize(16, 16))
        # Create "path" line edit
        self._line = QtWidgets.QLineEdit(self)
        self._line.setReadOnly(True)
        self._line.setStyleSheet("QLineEdit { background:#ddd; }")
        self._line.setFocusPolicy(QtCore.Qt.NoFocus)
        # Create options menu
        self._options = QtWidgets.QToolButton(self)
        self._options.setIcon(pyzo.icons.filter)
        self._options.setIconSize(QtCore.QSize(16, 16))
        self._options.setPopupMode(self._options.InstantPopup)
        self._options.setToolButtonStyle(QtCore.Qt.ToolButtonTextBesideIcon)
        #
        self._options._menu = QtWidgets.QMenu()
        self._options.setMenu(self._options._menu)
        self.onOptionsPress()  # create menu now
        # Create tree
        self._tree = WorkspaceTree(self)
        # Create message for when tree is empty
        self._initText = QtWidgets.QLabel(
            pyzo.translate(
                "pyzoWorkspace",
                """Lists the variables in the current shell's namespace.
Currently, there are none. Some of them may be hidden because of the filters you
configured.""",
            ),
            self,
        )
        self._initText.setVisible(False)
        self._initText.setWordWrap(True)
        # Set layout
        layout = QtWidgets.QHBoxLayout()
        layout.addWidget(self._up, 0)
        layout.addWidget(self._line, 1)
        layout.addWidget(self._options, 0)
        #
        mainLayout = QtWidgets.QVBoxLayout(self)
        mainLayout.addLayout(layout, 0)
        mainLayout.addWidget(self._initText, 1)
        mainLayout.addWidget(self._tree, 2)
        mainLayout.setSpacing(2)
        # set margins
        margin = pyzo.config.view.widgetMargin
        mainLayout.setContentsMargins(margin, margin, margin, margin)
        self.setLayout(mainLayout)
        # Bind events
        self._up.pressed.connect(self._tree._proxy.goUp)
        self._options.pressed.connect(self.onOptionsPress)
        self._options._menu.triggered.connect(self.onOptionMenuTiggered)
    def displayEmptyWorkspace(self, empty):
        self._tree.setVisible(not empty)
        self._initText.setVisible(empty)
    def onOptionsPress(self):
        """Create the menu for the button, Do each time to make sure
        the checks are right."""
        # Get menu
        menu = self._options._menu
        menu.clear()
        hideables = [
            ("type", pyzo.translate("pyzoWorkspace", "Hide types")),
            ("function", pyzo.translate("pyzoWorkspace", "Hide functions")),
            ("module", pyzo.translate("pyzoWorkspace", "Hide modules")),
            ("private", pyzo.translate("pyzoWorkspace", "Hide private
identifiers")),
            (
                "startup",
                pyzo.translate("pyzoWorkspace", "Hide the shell's startup
variables"),
            ),
        ]
        for type, display in hideables:
            checked = type in self._config.hideTypes
            action = menu.addAction(display)
            action._what = type
            action.setCheckable(True)
            action.setChecked(checked)
    def onOptionMenuTiggered(self, action):
        """The user decides what to hide in the workspace."""
        # What to show
        type = action._what.lower()
        # Swap
        if type in self._config.hideTypes:
            while type in self._config.hideTypes:
                self._config.hideTypes.remove(type)
        else:
            self._config.hideTypes.append(type)
        # Update
        self._tree.fillWorkspace()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Package tools of pyzo
A tool consists of a module which contains a class. The id of
a tool is its module name made lower case. The module should
contain a class corresponding to its id. We advise to follow the
common python style and start the class name with a capital
letter, case does not matter for the tool to work though.
For instance, the tool "pyzologger" is the class "PyzoLogger" found
in module "pyzoLogger"
The module may contain the following extra variables (which should
be placed within the first 50 lines of code):
tool_name - A readable name for the tool (may contain spaces,
will be shown in the tab)
tool_summary - A single line short summary of the tool. To be
displayed in the statusbar.
"""
# tools I'd like:
# - find in files
# - workspace
# - source tree
# - snipet manager
# - file browser
# - pythonpath editor, startupfile editor (or as part of pyzo?)
import os, sys, imp
import pyzo
from pyzo.qt import QtCore, QtGui, QtWidgets  # noqa
from pyzo.util import zon as ssdf
from pyzo import translate  # noqa (we have an eval down here)
class ToolDockWidget(QtWidgets.QDockWidget):
    """A dock widget that holds a tool.
    It sets all settings, initializes the tool widget, and notifies the
    tool manager on closing.
    """
    def __init__(self, parent, toolManager):
        QtWidgets.QDockWidget.__init__(self, parent)
        # Store stuff
        self._toolManager = toolManager
        # Allow docking anywhere, othwerise restoring state wont work properly
        # Set other settings
        self.setFeatures(
            QtWidgets.QDockWidget.DockWidgetMovable
            | QtWidgets.QDockWidget.DockWidgetClosable
            | QtWidgets.QDockWidget.DockWidgetFloatable
            # QtWidgets.QDockWidget.DockWidgetVerticalTitleBar
        )
    def setTool(self, toolId, toolName, toolClass):
        """Set the tool information. Call this right after
        initialization."""
        # Store id and set object name to enable saving/restoring state
        self._toolId = toolId
        self.setObjectName(toolId)
        # Set name
        self.setWindowTitle(toolName)
        # Create tool widget
        self.reload(toolClass)
    def closeEvent(self, event):
        if self._toolManager:
            self._toolManager.onToolClose(self._toolId)
            self._toolManager = None
        # Close and delete widget
        old = self.widget()
        if old:
            old.close()
            old.deleteLater()
        # Close and delete dock widget
        self.close()
        self.deleteLater()
        # We handled the event
        event.accept()
    def reload(self, toolClass):
        """Reload the widget with a new widget class."""
        old = self.widget()
        new = toolClass(pyzo.main)
        self.setWidget(new)
        if old:
            old.close()
            old.deleteLater()
class ToolDescription:
    """Provides a description of a tool and has a reference to
    the tool dock instance if it is loaded.
    """
    def __init__(self, modulePath, name="", description=""):
        # Set names
        self.modulePath = modulePath
        self.moduleName = os.path.splitext(os.path.basename(modulePath))[0]
        self.id = self.moduleName.lower()
        if name:
            self.name = name
        else:
            self.name = self.id
        # Set description
        self.description = description
        # Init instance to None, will be set when loaded
        self.instance = None
    def menuLauncher(self, value):
        """Function that is called by the menu when this tool is selected."""
        if value is None:
            return bool(self.instance)
            # return self.id in pyzo.toolManager._activeTools
        elif value:
            pyzo.toolManager.loadTool(self.id)
        else:
            self.widget = None
            pyzo.toolManager.closeTool(self.id)
class ToolManager(QtCore.QObject):
    """Manages the tools."""
    # This signal indicates a change in the loaded tools
    toolInstanceChange = QtCore.Signal()
    def __init__(self, parent=None):
        QtCore.QObject.__init__(self, parent)
        # list of ToolDescription instances
        self._toolInfo = None
        self._activeTools = {}
    def loadToolInfo(self):
        """(re)load the tool information."""
        # Get paths to load files from
        toolDir1 = os.path.join(pyzo.pyzoDir, "tools")
        toolDir2 = os.path.join(pyzo.appDataDir, "tools")
        # Create list of tool files
        toolfiles = []
        for toolDir in [toolDir1, toolDir2]:
            tmp = [os.path.join(toolDir, f) for f in os.listdir(toolDir)]
            toolfiles.extend(tmp)
        # Note: we do not use the code below anymore, since even the frozen
        # app makes use of the .py files.
        #         # Get list of files, also when we're in a zip file.
        #         i = tooldir.find('.zip')
        #         if i>0:
        #             # Get list of files from zipfile
        #             tooldir = tooldir[:i+4]
        #             import zipfile
        #             z = zipfile.ZipFile(tooldir)
        #             toolfiles = [os.path.split(i)[1] for i in z.namelist()
        #                         if i.startswith('visvis') and
i.count('functions')]
        #         else:
        #             # Get list of files from file system
        #             toolfiles = os.listdir(tooldir)
        # Iterate over tool modules
        newlist = []
        for file in toolfiles:
            modulePath = file
            # Check
            if os.path.isdir(file):
                file = os.path.join(file, "__init__.py")  # A package perhaps
                if not os.path.isfile(file):
                    continue
            elif file.endswith("__.py") or not file.endswith(".py"):
                continue
            elif file.endswith("pyzoFileBrowser.py"):
                # Skip old file browser (the file can be there from a previous
install)
                continue
            #
            toolName = ""
            toolSummary = ""
            # read file to find name or summary
            linecount = 0
            for line in open(file, encoding="utf-8"):
                linecount += 1
                if linecount > 50:
                    break
                if line.startswith("tool_name"):
                    i = line.find("=")
                    if i < 0:
                        continue
                    line = line.rstrip("\n").rstrip("\r")
                    line = line[i + 1 :].strip(" ")
                    toolName = eval(line)  # applies translation
                elif line.startswith("tool_summary"):
                    i = line.find("=")
                    if i < 0:
                        continue
                    line = line.rstrip("\n").rstrip("\r")
                    line = line[i + 1 :].strip(" ")
                    toolSummary = line.strip("'").strip('"')
                else:
                    pass
            # Add stuff
            tmp = ToolDescription(modulePath, toolName, toolSummary)
            newlist.append(tmp)
        # Store and return
        self._toolInfo = sorted(newlist, key=lambda x: x.id)
        self.updateToolInstances()
        return self._toolInfo
    def updateToolInstances(self):
        """Make tool instances up to date, so that it can be seen what
        tools are now active."""
        for toolDes in self.getToolInfo():
            if toolDes.id in self._activeTools:
                toolDes.instance = self._activeTools[toolDes.id]
            else:
                toolDes.instance = None
        # Emit update signal
        self.toolInstanceChange.emit()
    def getToolInfo(self):
        """Like loadToolInfo(), but use buffered instance if available."""
        if self._toolInfo is None:
            self.loadToolInfo()
        return self._toolInfo
    def getToolClass(self, toolId):
        """Get the class of the tool.
        It will import (and reload) the module and get the class.
        Some checks are performed, like whether the class inherits
        from QWidget.
        Returns the class or None if failed...
        """
        # Make sure we have the info
        if self._toolInfo is None:
            self.loadToolInfo()
        # Get module name and path
        for toolDes in self._toolInfo:
            if toolDes.id == toolId:
                moduleName = toolDes.moduleName
                modulePath = toolDes.modulePath
                break
        else:
            print("WARNING: could not find module for tool", repr(toolId))
            return None
        # Remove from sys.modules, to force the module to reload
        for key in [key for key in sys.modules]:
            if key and key.startswith("pyzo.tools." + moduleName):
                del sys.modules[key]
        # Load module
        try:
            m_file, m_fname, m_des = imp.find_module(
                moduleName, [os.path.dirname(modulePath)]
            )
            mod = imp.load_module("pyzo.tools." + moduleName, m_file, m_fname,
m_des)
        except Exception as why:
            print("Invalid tool " + toolId + ":", why)
            return None
        # Is the expected class present?
        className = ""
        for member in dir(mod):
            if member.lower() == toolId:
                className = member
                break
        else:
            print("Invalid tool, Classname must match module name '%s'!" %
toolId)
            return None
        # Does it inherit from QWidget?
        plug = mod.__dict__[className]
        if not (isinstance(plug, type) and issubclass(plug, QtWidgets.QWidget)):
            print("Invalid tool, tool class must inherit from QWidget!")
            return None
        # Succes!
        return plug
    def loadTool(self, toolId, splitWith=None):
        """Load a tool by creating a dock widget containing the tool widget."""
        # A tool id should always be lower case
        toolId = toolId.lower()
        # Close old one
        if toolId in self._activeTools:
            old = self._activeTools[toolId].widget()
            self._activeTools[toolId].setWidget(QtWidgets.QWidget(pyzo.main))
            if old:
                old.close()
                old.deleteLater()
        # Get tool class (returns None on failure)
        toolClass = self.getToolClass(toolId)
        if toolClass is None:
            return
        # Already loaded? reload!
        if toolId in self._activeTools:
            self._activeTools[toolId].reload(toolClass)
            return
        # Obtain name from buffered list of names
        for toolDes in self._toolInfo:
            if toolDes.id == toolId:
                name = toolDes.name
                break
        else:
            name = toolId
        # Make sure there is a config entry for this tool
        if not hasattr(pyzo.config.tools, toolId):
            pyzo.config.tools[toolId] = ssdf.new()
        # Create dock widget and add in the main window
        dock = ToolDockWidget(pyzo.main, self)
        dock.setTool(toolId, name, toolClass)
        if splitWith and splitWith in self._activeTools:
            otherDock = self._activeTools[splitWith]
            pyzo.main.splitDockWidget(otherDock, dock, QtCore.Qt.Horizontal)
        else:
            pyzo.main.addDockWidget(QtCore.Qt.RightDockWidgetArea, dock)
        # Add to list
        self._activeTools[toolId] = dock
        self.updateToolInstances()
    def reloadTools(self):
        """Reload all tools."""
        for id in self.getLoadedTools():
            self.loadTool(id)
    def closeTool(self, toolId):
        """Close the tool with specified id."""
        if toolId in self._activeTools:
            dock = self._activeTools[toolId]
            dock.close()
    def getTool(self, toolId):
        """Get the tool widget instance, or None
        if not available."""
        if toolId in self._activeTools:
            return self._activeTools[toolId].widget()
        else:
            return None
    def onToolClose(self, toolId):
        # Remove from dict
        self._activeTools.pop(toolId, None)
        # Set instance to None
        self.updateToolInstances()
    def getLoadedTools(self):
        """Get a list with id's of loaded tools."""
        tmp = []
        for toolDes in self.getToolInfo():
            if toolDes.id in self._activeTools:
                tmp.append(toolDes.id)
        return tmp
```

```
import sys
import os.path as op
import pyzo
from pyzo import translate
from pyzo.util import zon as ssdf
from . import QtCore, QtGui, QtWidgets
from . import proxies
from .tree import Tree
from .utils import cleanpath, isdir
class Browser(QtWidgets.QWidget):
    """A browser consists of an address bar, and tree view, and other
    widets to help browse the file system. The browser object is responsible
    for tying the different browser-components together.
    It is also provides the API for dealing with starred dirs.
    """
    def __init__(self, parent, config, path=None):
        QtWidgets.QWidget.__init__(self, parent)
        # Store config
        self.config = config
        # Create star button
        self._projects = Projects(self)
        # Create path input/display lineEdit
        self._pathEdit = PathInput(self)
        # Create file system proxy
        self._fsProxy = proxies.NativeFSProxy()
        self.destroyed.connect(self._fsProxy.stop)
        # Create tree widget
        self._tree = Tree(self)
        self._tree.setPath(cleanpath(self.config.path))
        # Create name filter
        self._nameFilter = NameFilter(self)
        # self._nameFilter.lineEdit().setToolTip('File filter pattern')
        self._nameFilter.setToolTip(translate("filebrowser", "Filename filter"))
        self._nameFilter.setPlaceholderText(self._nameFilter.toolTip())
        # Create search filter
        self._searchFilter = SearchFilter(self)
        self._searchFilter.setToolTip(translate("filebrowser", "Search in
files"))
        self._searchFilter.setPlaceholderText(self._searchFilter.toolTip())
        # Signals to sync path.
        # Widgets that can change the path transmit signal to _tree
        self._pathEdit.dirUp.connect(self._tree.setFocus)
        self._pathEdit.dirUp.connect(self._tree.setPathUp)
        self._pathEdit.dirChanged.connect(self._tree.setPath)
        self._projects.dirChanged.connect(self._tree.setPath)
        #
        self._nameFilter.filterChanged.connect(self._tree.onChanged)  # ==
update
        self._searchFilter.filterChanged.connect(self._tree.onChanged)
        # The tree transmits signals to widgets that need to know the path
        self._tree.dirChanged.connect(self._pathEdit.setPath)
        self._tree.dirChanged.connect(self._projects.setPath)
        self._layout()
        # Set and sync path ...
        if path is not None:
            self._tree.SetPath(path)
        self._tree.dirChanged.emit(self._tree.path())
    def getImportWizard(self):
        # Lazy loading
        try:
            return self._importWizard
        except AttributeError:
            from .importwizard import ImportWizard
            self._importWizard = ImportWizard()
            return self._importWizard
    def _layout(self):
        layout = QtWidgets.QVBoxLayout(self)
        layout.setContentsMargins(0, 0, 0, 0)
        # layout.setSpacing(6)
        self.setLayout(layout)
        #
        layout.addWidget(self._projects)
        layout.addWidget(self._pathEdit)
        layout.addWidget(self._tree)
        #
        subLayout = QtWidgets.QHBoxLayout()
        subLayout.setSpacing(2)
        subLayout.addWidget(self._nameFilter, 5)
        subLayout.addWidget(self._searchFilter, 5)
        layout.addLayout(subLayout)
    def cleanUp(self):
        self._fsProxy.stop()
    def nameFilter(self):
        # return self._nameFilter.lineEdit().text()
        return self._nameFilter.text()
    def searchFilter(self):
        return {
            "pattern": self._searchFilter.text(),
            "matchCase": self.config.searchMatchCase,
            "regExp": self.config.searchRegExp,
            "subDirs": self.config.searchSubDirs,
        }
    @property
    def expandedDirs(self):
        """The list of the expanded directories."""
        return self.parent().config.expandedDirs
    @property
    def starredDirs(self):
        """A list of the starred directories."""
        return [d.path for d in self.parent().config.starredDirs]
    def dictForStarredDir(self, path):
        """Return the dict of the starred dir corresponding to
        the given path, or None if no starred dir was found.
        """
        if not path:
            return None
        for d in self.parent().config.starredDirs:
            if op.normcase(d["path"]) == op.normcase(path):
                return d
        else:
            return None
    def addStarredDir(self, path):
        """Add the given path to the starred directories."""
        # Create new dict
        newProject = ssdf.new()
        newProject.path = op.normcase(path)  # Normalize case!
        newProject.name = op.basename(path)
        newProject.addToPythonpath = False
        # Add it to the config
        self.parent().config.starredDirs.append(newProject)
        # Update list
        self._projects.updateProjectList()
    def removeStarredDir(self, path):
        """Remove the given path from the starred directories.
        The path must exactlty match.
        """
        # Remove
        starredDirs = self.parent().config.starredDirs
        pathn = op.normcase(path)
        for d in starredDirs:
            if op.normcase(pathn) == op.normcase(d.path):
                starredDirs.remove(d)
        # Update list
        self._projects.updateProjectList()
    def test(self, sort=False):
        items = []
        for i in range(self._tree.topLevelItemCount()):
            item = self._tree.topLevelItem(i)
            items.append(item)
            # self._tree.removeItemWidget(item, 0)
        self._tree.clear()
        # items.sort(key=lambda x: x._path)
        items = [item for item in reversed(items)]
        for item in items:
            self._tree.addTopLevelItem(item)
    def currentProject(self):
        """Return the ssdf dict for the current project, or None."""
        return self._projects.currentDict()
class LineEditWithToolButtons(QtWidgets.QLineEdit):
    """Line edit to which tool buttons (with icons) can be attached."""
    def __init__(self, parent):
        QtWidgets.QLineEdit.__init__(self, parent)
        self._leftButtons = []
        self._rightButtons = []
    def addButtonLeft(self, icon, willHaveMenu=False):
        return self._addButton(icon, willHaveMenu, self._leftButtons)
    def addButtonRight(self, icon, willHaveMenu=False):
        return self._addButton(icon, willHaveMenu, self._rightButtons)
    def _addButton(self, icon, willHaveMenu, L):
        # Create button
        button = QtWidgets.QToolButton(self)
        L.append(button)
        # Customize appearance
        button.setIcon(icon)
        button.setIconSize(QtCore.QSize(16, 16))
        button.setStyleSheet("QToolButton { border: none; padding: 0px; }")
        # button.setStyleSheet("QToolButton { border: none; padding: 0px;
background-color:red;}");
        # Set behavior
        button.setCursor(QtCore.Qt.ArrowCursor)
        button.setPopupMode(button.InstantPopup)
        # Customize alignment
        if willHaveMenu:
            button.setToolButtonStyle(QtCore.Qt.ToolButtonTextBesideIcon)
            if sys.platform.startswith("win"):
                button.setText(" ")
        # Update self
        self._updateGeometry()
        return button
    def setButtonVisible(self, button, visible):
        for but in self._leftButtons:
            if but is button:
                but.setVisible(visible)
        for but in self._rightButtons:
            if but is button:
                but.setVisible(visible)
        self._updateGeometry()
    def resizeEvent(self, event):
        QtWidgets.QLineEdit.resizeEvent(self, event)
        self._updateGeometry(True)
    def showEvent(self, event):
        QtWidgets.QLineEdit.showEvent(self, event)
        self._updateGeometry()
    def _updateGeometry(self, light=False):
        if not self.isVisible():
            return
        # Init
        rect = self.rect()
        # Determine padding and height
        paddingLeft, paddingRight, height = 1, 1, 0
        #
        for but in self._leftButtons:
            if but.isVisible():
                sz = but.sizeHint()
                height = max(height, sz.height())
                but.move(
                    int(1 + paddingLeft),
                    int(rect.bottom() + 1 - sz.height()) // 2,
                )
                paddingLeft += sz.width() + 1
        #
        for but in self._rightButtons:
            if but.isVisible():
                sz = but.sizeHint()
                paddingRight += sz.width() + 1
                height = max(height, sz.height())
                but.move(
                    int(rect.right() - 1 - paddingRight),
                    int(rect.bottom() + 1 - sz.height()) // 2,
                )
        # Set padding
        ss = "QLineEdit { padding-left: %ipx; padding-right: %ipx} "
        self.setStyleSheet(ss % (paddingLeft, paddingRight))
        # Set minimum size
        if not light:
            fw = QtWidgets.qApp.style().pixelMetric(
                QtWidgets.QStyle.PM_DefaultFrameWidth
            )
            msz = self.minimumSizeHint()
            w = max(msz.width(), paddingLeft + paddingRight + 10)
            h = max(msz.height(), height + fw * 2 + 2)
            self.setMinimumSize(w, h)
class PathInput(LineEditWithToolButtons):
    """Line edit for selecting a path."""
    dirChanged = QtCore.Signal(
        str
    )  # Emitted when the user changes the path (and is valid)
    dirUp = QtCore.Signal()  # Emitted when user presses the up button
    def __init__(self, parent):
        LineEditWithToolButtons.__init__(self, parent)
        # Create up button
        self._upBut = self.addButtonLeft(pyzo.icons.folder_parent)
        self._upBut.clicked.connect(self.dirUp)
        # To receive focus events
        self.setFocusPolicy(QtCore.Qt.StrongFocus)
        # Set completion mode
        self.setCompleter(QtWidgets.QCompleter())
        c = self.completer()
        c.setCompletionMode(c.InlineCompletion)
        # Set dir model to completer
        dirModel = QtWidgets.QFileSystemModel(c)
        dirModel.setFilter(QtCore.QDir.Dirs | QtCore.QDir.NoDotAndDotDot)
        c.setModel(dirModel)
        # Connect signals
        # c.activated.connect(self.onActivated)
        self.textEdited.connect(self.onTextEdited)
        # self.textChanged.connect(self.onTextEdited)
        # self.cursorPositionChanged.connect(self.onTextEdited)
    def setPath(self, path):
        """Set the path to display. Does nothing if this widget has focus."""
        if not self.hasFocus():
            self.setText(path)
            self.checkValid()  # Reset style if it was invalid first
    def checkValid(self):
        # todo: This kind of violates the abstraction of the file system
        # ok for now, but we should find a different approach someday
        # Check
        text = self.text()
        dir = cleanpath(text)
        isvalid = text and isdir(dir) and op.isabs(dir)
        # Apply styling
        ss = self.styleSheet().replace("font-style:italic; ", "")
        if not isvalid:
            ss = ss.replace("QLineEdit {", "QLineEdit {font-style:italic; ")
        self.setStyleSheet(ss)
        # Return
        return isvalid
    def event(self, event):
        # Capture key events to explicitly apply the completion and
        # invoke checking whether the current text is a valid directory.
        # Test if QtGui is not None (can happen when reloading tools)
        if QtGui and isinstance(event, QtGui.QKeyEvent):
            qt = QtCore.Qt
            if event.key() in [qt.Key_Tab, qt.Key_Enter, qt.Key_Return]:
                self.setText(self.text())  # Apply completion
                self.onTextEdited()  # Check if this is a valid dir
                return True
        return super().event(event)
    def onTextEdited(self, dummy=None):
        text = self.text()
        if self.checkValid():
            self.dirChanged.emit(cleanpath(text))
    def focusOutEvent(self, event=None):
        """focusOutEvent(event)
        On focusing out, make sure that the set path is correct.
        """
        if event is not None:
            QtWidgets.QLineEdit.focusOutEvent(self, event)
        path = self.parent()._tree.path()
        self.setPath(path)
class Projects(QtWidgets.QWidget):
    dirChanged = QtCore.Signal(str)  # Emitted when the user changes the project
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        # Init variables
        self._path = ""
        # Create combo button
        self._combo = QtWidgets.QComboBox(self)
        self._combo.setEditable(False)
        self.updateProjectList()
        # Create star button
        self._but = QtWidgets.QToolButton(self)
        self._but.setIcon(pyzo.icons.star3)
        self._but.setStyleSheet("QToolButton { padding: 0px; }")
        self._but.setIconSize(QtCore.QSize(18, 18))
        self._but.setToolButtonStyle(QtCore.Qt.ToolButtonTextBesideIcon)
        self._but.setPopupMode(self._but.InstantPopup)
        #
        self._menu = QtWidgets.QMenu(self._but)
        self._menu.triggered.connect(self.onMenuTriggered)
        self.buildMenu()
        # Make equal height
        h = max(self._combo.sizeHint().height(), self._but.sizeHint().height())
        self._combo.setMinimumHeight(h)
        self._but.setMinimumHeight(h)
        # Connect signals
        self._but.pressed.connect(self.onButtonPressed)
        self._combo.activated.connect(self.onProjectSelect)
        # Layout
        layout = QtWidgets.QHBoxLayout(self)
        self.setLayout(layout)
        layout.addWidget(self._but)
        layout.addWidget(self._combo)
        layout.setSpacing(2)
        layout.setContentsMargins(0, 0, 0, 0)
    def currentDict(self):
        """Return the current project-dict, or None."""
        path = self._combo.itemData(self._combo.currentIndex())
        return self.parent().dictForStarredDir(path)
    def setPath(self, path):
        self._path = path
        # Find project index
        projectIndex, L = 0, 0
        pathn = op.normcase(path) + op.sep
        for i in range(self._combo.count()):
            projectPath = self._combo.itemData(i) + op.sep
            if pathn.startswith(projectPath) and len(projectPath) > L:
                projectIndex, L = i, len(projectPath)
        # Select project or not ...
        self._combo.setCurrentIndex(projectIndex)
        if projectIndex:
            self._but.setIcon(pyzo.icons.star2)
            self._but.setMenu(self._menu)
        else:
            self._but.setIcon(pyzo.icons.star3)
            self._but.setMenu(None)
    def updateProjectList(self):
        # Get sorted version of starredDirs
        starredDirs = self.parent().starredDirs
        starredDirs.sort(key=lambda p:
self.parent().dictForStarredDir(p).name.lower())
        # Refill the combo box
        self._combo.clear()
        if starredDirs:
            self._combo.addItem(
                translate("filebrowser", "Projects:"), ""
            )  # No-project item
            for p in starredDirs:
                name = self.parent().dictForStarredDir(p).name
                self._combo.addItem(name, p)
        else:
            self._combo.addItem(
                translate("filebrowser", "Click star to bookmark current dir"),
""
            )
    def buildMenu(self):
        menu = self._menu
        menu.clear()
        # Add action to remove bookmark
        action = menu.addAction(translate("filebrowser", "Remove project"))
        action._id = "remove"
        action.setCheckable(False)
        # Add action to change name
        action = menu.addAction(translate("filebrowser", "Change project name"))
        action._id = "name"
        action.setCheckable(False)
        menu.addSeparator()
        # Add check action for adding to Pythonpath
        action = menu.addAction(translate("filebrowser", "Add path to Python
path"))
        action._id = "pythonpath"
        action.setCheckable(True)
        d = self.currentDict()
        if d:
            checked = bool(d and d["addToPythonpath"])
            action.setChecked(checked)
        # Add action to cd to the project directory
        action = menu.addAction(
            translate("filebrowser", "Go to this directory in the current
shell")
        )
        action._id = "cd"
        action.setCheckable(False)
    def onMenuTriggered(self, action):
        d = self.currentDict()
        if not d:
            return
        if action._id == "remove":
            # Remove this project
            self.parent().removeStarredDir(d.path)
        elif action._id == "name":
            # Open dialog to ask for name
            name = QtWidgets.QInputDialog.getText(
                self.parent(),
                translate("filebrowser", "Project name"),
                translate("filebrowser", "New project name:"),
                text=d["name"],
            )
            if isinstance(name, tuple):
                name = name[0] if name[1] else ""
            if name:
                d["name"] = name
            self.updateProjectList()
        elif action._id == "pythonpath":
            # Flip add-to-pythonpath flag
            d["addToPythonpath"] = not d["addToPythonpath"]
        elif action._id == "cd":
            # cd to the directory
            shell = pyzo.shells.getCurrentShell()
            if shell:
                shell.executeCommand("cd " + d.path + "\n")
    def onButtonPressed(self):
        if self._but.menu():
            # The directory is starred and has a menu. The user just
            # used the menu (or not). Update so it is up-to-date next time.
            self.buildMenu()
        else:
            # Not starred right now, create new project!
            self.parent().addStarredDir(self._path)
        # Update
        self.setPath(self._path)
    def onProjectSelect(self, index):
        path = self._combo.itemData(index)
        if path:
            # Go to dir
            self.dirChanged.emit(path)
        else:
            # Dummy item, reset
            self.setPath(self._path)
class NameFilter(LineEditWithToolButtons):
    """Combobox to filter by name."""
    filterChanged = QtCore.Signal()
    def __init__(self, parent):
        LineEditWithToolButtons.__init__(self, parent)
        # Create tool button, and attach the menu
        self._menuBut = self.addButtonRight(pyzo.icons["filter"], True)
        self._menu = QtWidgets.QMenu(self._menuBut)
        self._menu.triggered.connect(self.onMenuTriggered)
        self._menuBut.setMenu(self._menu)
        #
        # Add common patterns
        for pattern in [
            "*",
            "!hidden",
            "!*.pyc !hidden",
            "*.py *.pyw",
            "*.py *.pyw *.pyx *.pxd",
            "*.h *.c *.cpp",
        ]:
            self._menu.addAction(pattern)
        # Emit signal when value is changed
        self._lastValue = ""
        self.returnPressed.connect(self.checkFilterValue)
        self.editingFinished.connect(self.checkFilterValue)
        # Ensure the namefilter is in the config and initialize
        config = self.parent().config
        if "nameFilter" not in config:
            config.nameFilter = "!*.pyc"
        self.setText(config.nameFilter)
    def setText(self, value, test=False):
        """To initialize the name filter."""
        QtWidgets.QLineEdit.setText(self, value)
        if test:
            self.checkFilterValue()
        self._lastValue = value
    def checkFilterValue(self):
        value = self.text()
        if value != self._lastValue:
            self.parent().config.nameFilter = value
            self._lastValue = value
            self.filterChanged.emit()
    def onMenuTriggered(self, action):
        self.setText(action.text(), True)
class SearchFilter(LineEditWithToolButtons):
    """Line edit to do a search in the files."""
    filterChanged = QtCore.Signal()
    def __init__(self, parent):
        LineEditWithToolButtons.__init__(self, parent)
        # Create tool button, and attach the menu
        self._menuBut = self.addButtonRight(pyzo.icons["magnifier"], True)
        self._menu = QtWidgets.QMenu(self._menuBut)
        self._menu.triggered.connect(self.onMenuTriggered)
        self._menuBut.setMenu(self._menu)
        self.buildMenu()
        # Create cancel button
        self._cancelBut = self.addButtonRight(pyzo.icons["cancel"])
        self._cancelBut.setVisible(False)
        # Keep track of last value of search (initialized empty)
        self._lastValue = ""
        # Connect signals
        self._cancelBut.pressed.connect(self.onCancelPressed)
        self.textChanged.connect(self.updateCancelButton)
        self.editingFinished.connect(self.checkFilterValue)
        self.returnPressed.connect(self.forceFilterChanged)
    def onCancelPressed(self):
        """Clear text or build menu."""
        if self.text():
            QtWidgets.QLineEdit.clear(self)
            self.checkFilterValue()
        else:
            self.buildMenu()
    def checkFilterValue(self):
        value = self.text()
        if value != self._lastValue:
            self._lastValue = value
            self.filterChanged.emit()
    def forceFilterChanged(self):
        self._lastValue = self.text()
        self.filterChanged.emit()
    def updateCancelButton(self, text):
        visible = bool(self.text())
        self.setButtonVisible(self._cancelBut, visible)
    def buildMenu(self):
        config = self.parent().config
        menu = self._menu
        menu.clear()
        map = [
            ("searchMatchCase", False, translate("filebrowser", "Match case")),
            ("searchRegExp", False, translate("filebrowser", "RegExp")),
            ("searchSubDirs", True, translate("filebrowser", "Search in 
subdirs")),
        ]
        # Fill menu
        for option, default, description in map:
            if option is None:
                menu.addSeparator()
            else:
                # Make sure the option exists
                if option not in config:
                    config[option] = default
                # Make action in menu
                action = menu.addAction(description)
                action._option = option
                action.setCheckable(True)
                action.setChecked(bool(config[option]))
    def onMenuTriggered(self, action):
        config = self.parent().config
        option = action._option
        # Swap this option
        if option in config:
            config[option] = not config[option]
        else:
            config[option] = True
        # Update
        self.filterChanged.emit()
```

```
"""
The import wizard helps the user importing CSV-like data from a file into a
numpy array. The wizard containst three pages:
SelectFilePage:
    - The user selects a file and previews its contents (or, the beginning of
it)
SetParametersPage:
    - The user selects delimiters, etc. and selects which columns to import
    - A preview of the data in tabualar form is shown, with colors indicating
      how the file is parsed: yellow for header rows, green for the comments
      column and red for values that could not be parsed
ResultPage:
    - The wizard shows the generated code that is to be used to import the file
      according to the settings
    - The user chooses to execute the code in the current shell or paste the
      code into the editor
"""
import unicodedata
import os.path as op
import pyzo.codeeditor
from . import QtCore, QtGui, QtWidgets
from pyzo import translate
# All keywords in Python 2 and 3. Obtained using: import keyword; keyword.kwlist
# Merged from Py2 and 3
keywords = [
    "False",
    "None",
    "True",
    "and",
    "as",
    "assert",
    "break",
    "class",
    "continue",
    "def",
    "del",
    "elif",
    "else",
    "except",
    "exec",
    "finally",
    "for",
    "from",
    "global",
    "if",
    "import",
    "in",
    "is",
    "lambda",
    "nonlocal",
    "not",
    "or",
    "pass",
    "print",
    "raise",
    "return",
    "try",
    "while",
    "with",
    "yield",
]
class CodeView(
    pyzo.codeeditor.IndentationGuides,
    pyzo.codeeditor.CodeFolding,
    pyzo.codeeditor.Indentation,
    pyzo.codeeditor.HomeKey,
    pyzo.codeeditor.EndKey,
    pyzo.codeeditor.NumpadPeriodKey,
    pyzo.codeeditor.AutoIndent,
    pyzo.codeeditor.PythonAutoIndent,
    pyzo.codeeditor.SyntaxHighlighting,
    pyzo.codeeditor.CodeEditorBase,
):  # CodeEditorBase must be the last one in the list
    """
    Code viewer, stripped down version of the CodeEditor
    """
    pass
class SelectFilePage(QtWidgets.QWizardPage):
    """
    First page of the wizard, select file and preview contents
    """
    def __init__(self):
        QtWidgets.QWizardPage.__init__(self)
        self.setTitle(translate("importwizard", "Select file"))
        self.txtFilename = QtWidgets.QLineEdit()
        self.btnBrowse = QtWidgets.QPushButton(translate("importwizard",
"Browse..."))
        self.preview = QtWidgets.QPlainTextEdit()
        self.preview.setReadOnly(True)
        vlayout = QtWidgets.QVBoxLayout()
        hlayout = QtWidgets.QHBoxLayout()
        hlayout.addWidget(self.txtFilename)
        hlayout.addWidget(self.btnBrowse)
        vlayout.addLayout(hlayout)
        vlayout.addWidget(QtWidgets.QLabel(translate("importwizard", 
"Preview:")))
        vlayout.addWidget(self.preview)
        self.setLayout(vlayout)
        self.registerField("fname", self.txtFilename)
        self.btnBrowse.clicked.connect(self.onBrowseClicked)
        self.txtFilename.editingFinished.connect(self.updatePreview)
        self._isComplete = False
    def onBrowseClicked(self):
        # Difference between PyQt4 and PySide: PySide returns filename, filter
        # while PyQt4 returns only the filename
        filename = QtWidgets.QFileDialog.getOpenFileName(
            filter="Text files (*.txt *.csv);;All files (*.*)"
        )
        if isinstance(filename, tuple):
            filename = filename[0]
        filename = str(filename).replace("/", op.sep)  # Show native file 
separator
        self.txtFilename.setText(filename)
        self.updatePreview()
    def updatePreview(self):
        filename = self.txtFilename.text()
        if not filename:
            data = ""
            self._isComplete = False
            self.wizard().setPreviewData(None)
        else:
            try:
                with open(filename, "rb") as file:
                    maxsize = 5000
                    data = file.read(maxsize)
                    more = bool(file.read(1))  # See if there is more data 
available
                data = data.decode("ascii", "replace")
                self.wizard().setPreviewData(data)
                if more:
                    data += "..."
                self._isComplete = True  # Allow to proceed to the next page
            except Exception as e:
                data = str(e)
                self._isComplete = False
                self.wizard().setPreviewData(None)
        self.preview.setPlainText(data)
        self.completeChanged.emit()
    def isComplete(self):
        return self._isComplete
class SetParametersPage(QtWidgets.QWizardPage):
    def __init__(self):
        QtWidgets.QWizardPage.__init__(self)
        self.setTitle("Select parameters")
        self._columnNames = None
        def genComboBox(choices):
            cbx = QtWidgets.QComboBox()
            for choice in choices:
                cbx.addItem(choice)
            cbx.setEditable(True)
            return cbx
        self.cbxDelimiter = genComboBox(",;")
        self.cbxComments = genComboBox("#%'")
        self.sbSkipHeader = QtWidgets.QSpinBox()
        self.preview = QtWidgets.QTableWidget()
        self.preview.setSelectionModel(
            QtCore.QItemSelectionModel(self.preview.model())
        )  # Work-around for reference tracking bug in PySide
        self.preview.setSelectionBehavior(self.preview.SelectColumns)
        self.preview.setSelectionMode(self.preview.MultiSelection)
        # Layout
        formlayout = QtWidgets.QFormLayout()
        formlayout.addRow("Delimiter", self.cbxDelimiter)
        formlayout.addRow("Comments", self.cbxComments)
        formlayout.addRow("Header rows to skip", self.sbSkipHeader)
        layout = QtWidgets.QVBoxLayout()
        layout.addLayout(formlayout)
        layout.addWidget(
            QtWidgets.QLabel(translate("importwizard", "Select columns to
import:"))
        )
        layout.addWidget(self.preview)
        self.setLayout(layout)
        # Wizard fields
        self.registerField("delimiter", self.cbxDelimiter, "currentText")
        self.registerField("comments", self.cbxComments, "currentText")
        self.registerField("skip_header", self.sbSkipHeader)
        # Signals
        self.cbxComments.editTextChanged.connect(self.updatePreview)
        self.cbxDelimiter.editTextChanged.connect(self.updatePreview)
        self.sbSkipHeader.valueChanged.connect(self.updatePreview)
self.preview.verticalHeader().sectionClicked.connect(self.onRowHeaderClicked)
    def columnNames(self):
        if self._columnNames is None:
            return list(
                ["d" + str(i + 1) for i in range(self.preview.columnCount() -
1)]
            )
        return list(self._columnNames)
    def updateHorizontalHeaderLabels(self):
        self.preview.setHorizontalHeaderLabels(self.columnNames() +
["Comments"])
    def onRowHeaderClicked(self, row):
        names = self.parseColumnNames(row)
        self._columnNames = names
        self.updateHorizontalHeaderLabels()
    def parseColumnNames(self, row):
        """
        Use the data in the given row to create column names. First, try the
        data in the data columns. If these are all empty, use the comments
        column, split by the given delimiter.
        Names are fixed up to be valid Python 2 / Python 3 identifiers
        (chars a-z A-Z _ 0-9 , no Python 2 or 3 keywords, not starting with 0-9)
        returns: list of names, exactly as many as there are data columns
        """
        names = []
        columnCount = self.preview.columnCount() - 1
        for col in range(columnCount):
            cell = self.preview.item(row, col)
            if cell is None:
                names.append("")
            else:
                names.append(cell.text().strip())
        # If no values found, try the comments:
        if not any(names):
            cell = self.preview.item(row, columnCount)
            if cell is not None:
                comment = cell.text()[1:].strip()  # Remove comment char and 
whitespace
                delimiter = self.cbxDelimiter.currentText()
                names = list(name.strip() for name in comment.split(delimiter))
                # Ensure names is exactly columnCount long
                names += [""] * columnCount
                names = names[:columnCount]
        # Fixup names
        def fixname(name, col):
            # Remove accents
            name = "".join(
                c
                for c in unicodedata.normalize("NFD", name)
                if unicodedata.category(c) != "Mn"
            )
            # Replace invalid chars with _
            name = "".join(
                c
                if (c.lower() >= "a" and c.lower() <= "z") or (c >= "0" and c <= 
"9")
                else "_"
                for c in name
            )
            if not name:
                return "d" + str(col)
            if name[0] >= "0" and name <= "9":
                name = "d" + name
            if name in keywords:
                name = name + "_"
            return name
        names = list(fixname(name, i + 1) for i, name in enumerate(names))
        return names
    def selectedColumns(self):
        """
        Returns a tuple of the columns that are selected, or None if no columns
        are selected
        """
        selected = []
        for selrange in self.preview.selectionModel().selection():
            selected += range(selrange.left(), selrange.right() + 1)
        selected.sort()
        if not selected:
            return None
        else:
            return tuple(selected)
    def initializePage(self):
        self.updatePreview()
    def updatePreview(self):
        # Get settings from the currently specified values in the wizard
        comments = self.cbxComments.currentText()
        delimiter = self.cbxDelimiter.currentText()
        skipheader = self.sbSkipHeader.value()
        if not comments or not delimiter:
            return
        # Store current selection, will be restored at the end
        selectedColumns = self.selectedColumns()
        # Clear the complete table
        self.preview.clear()
        self.preview.setColumnCount(0)
        self.preview.setRowCount(0)
        # Iterate through the source file line by line
        # Process like genfromtxt, with names = False
        # However, we do keep the header lines and comments; we show them in
        # distinct colors so that the user can see how the data is selected
        source = iter(self.wizard().previewData().splitlines())
        def split_line(line):
            """Chop off comments, strip, and split at delimiter."""
            line, sep, commentstr = line.partition(comments)
            line = line.strip(" \r\n")
            if line:
                return line.split(delimiter), sep + commentstr
            else:
                return [], sep + commentstr
        # Insert comments column
        self.preview.insertColumn(0)
        ncols = 0  # Number of columns, excluding comments column
        headerrows = 0  # Number of header rows, including empty header rows
        inheader = True
        for lineno, line in enumerate(source):
            fields, commentstr = split_line(line)
            # Process header like genfromtxt, with names = False
            if lineno >= skipheader and fields:
                inheader = False
            if inheader:
                headerrows = lineno + 1  # +1 since lineno = 0 is the first line
            self.preview.insertRow(lineno)
            # Add columns to fit all fields
            while len(fields) > ncols:
                self.preview.insertColumn(ncols)
                ncols += 1
            # Add fields to the table
            for col, field in enumerate(fields):
                cell = QtWidgets.QTableWidgetItem(field)
                if not inheader:
                    try:
                        float(field)
                    except ValueError:
                        cell.setBackground(QtGui.QBrush(QtGui.QColor("pink")))
                self.preview.setItem(lineno, col, cell)
            # Add the comment
            cell = QtWidgets.QTableWidgetItem(commentstr)
            cell.setBackground(QtGui.QBrush(QtGui.QColor("lightgreen")))
            self.preview.setItem(lineno, ncols, cell)
        # Colorize the header cells. This is done as the last step, since
        # meanwhile new columns (and thus new cells) may have been added
        for row in range(headerrows):
            for col in range(ncols):
                cell = self.preview.item(row, col)
                if not cell:
                    cell = QtWidgets.QTableWidgetItem("")
                    self.preview.setItem(row, col, cell)
                cell.setBackground(QtGui.QBrush(QtGui.QColor("khaki")))
        # Try to restore selection
        if selectedColumns is not None:
            for column in selectedColumns:
                self.preview.selectColumn(column)
        # Restore column names
        self.updateHorizontalHeaderLabels()
class ResultPage(QtWidgets.QWizardPage):
    """
    The resultpage lets the user select wether to import the data as a single
    2D-array, or as one variable (1D-array) per column
    Then, the code to do the import is generated (Py2 and Py3 compatible). This
    code can be executed in the current shell, or copied into the current editor
    """
    def __init__(self):
        QtWidgets.QWizardPage.__init__(self)
        self.setTitle("Execute import")
        self.setButtonText(
            QtWidgets.QWizard.FinishButton, translate("importwizard", "Close")
        )
        self.rbAsArray = QtWidgets.QRadioButton(
            translate("importwizard", "Import data as single array")
        )
        self.rbPerColumn = QtWidgets.QRadioButton(
            translate("importwizard", "Import data into one variable per
column")
        )
        self.rbAsArray.setChecked(True)
        self.chkInvalidRaise = QtWidgets.QCheckBox(
            translate("importwizard", "Raise error upon invalid data")
        )
        self.chkInvalidRaise.setChecked(True)
        self.codeView = CodeView()
        self.codeView.setParser("python")
        self.codeView.setZoom(pyzo.config.view.zoom)
        self.codeView.setFont(pyzo.config.view.fontname)
        self.btnExecute = QtWidgets.QPushButton("Execute in current shell")
        self.btnInsert = QtWidgets.QPushButton("Paste into current file")
        layout = QtWidgets.QVBoxLayout()
        layout.addWidget(self.rbAsArray)
        layout.addWidget(self.rbPerColumn)
        layout.addWidget(self.chkInvalidRaise)
        layout.addWidget(QtWidgets.QLabel("Resulting import code:"))
        layout.addWidget(self.codeView)
        layout.addWidget(self.btnExecute)
        layout.addWidget(self.btnInsert)
        self.setLayout(layout)
        self.registerField("invalid_raise", self.chkInvalidRaise)
        self.btnExecute.clicked.connect(self.onBtnExecuteClicked)
        self.btnInsert.clicked.connect(self.onBtnInsertClicked)
        self.rbAsArray.clicked.connect(self.updateCode)
        self.rbPerColumn.clicked.connect(self.updateCode)
        self.chkInvalidRaise.stateChanged.connect(lambda state:
self.updateCode())
    def initializePage(self):
        self.updateCode()
    def updateCode(self):
        perColumn = self.rbPerColumn.isChecked()
        if perColumn:
            columnNames = self.wizard().field("columnnames")
            usecols = self.wizard().field("usecols")
            if usecols is not None:  # User selected a subset of all columns
                # Pick corrsponding column names
                columnNames = [columnNames[i] for i in usecols]
            variables = ", ".join(columnNames)
        else:
            variables = "data"
        code = "import numpy\n"
        code += variables + " = numpy.genfromtxt(\n"
        for param, default in (
            ("fname", None),
            ("skip_header", 0),
            ("comments", "#"),
            ("delimiter", None),
            ("usecols", None),
            ("invalid_raise", True),
        ):
            value = self.wizard().field(param)
            if value != default:
                code += "\t%s = %r,\n" % (param, value)
        if perColumn:
            code += "\tunpack = True,\n"
        code += "\t)\n"
        self.codeView.setPlainText(code)
    def getCode(self):
        return self.codeView.toPlainText()
    def onBtnExecuteClicked(self):
        shell = pyzo.shells.getCurrentShell()
        if shell is None:
            QtWidgets.QMessageBox.information(
                self,
                translate("importwizard", "Import data wizard"),
                translate("importwizard", "No current shell active"),
            )
            return
        shell.executeCode(self.getCode(), "importwizard")
    def onBtnInsertClicked(self):
        editor = pyzo.editors.getCurrentEditor()
        if editor is None:
            QtWidgets.QMessageBox.information(
                self,
                translate("importwizard", "Import data wizard"),
                translate("importwizard", "No current file open"),
            )
            return
        code = self.getCode()
        # Format tabs/spaces according to editor setting
        if editor.indentUsingSpaces():
            code = code.replace("\t", " " * editor.indentWidth())
        # insert code at start of line
        cursor = editor.textCursor()
        cursor.movePosition(cursor.StartOfBlock)
        cursor.insertText(code)
class ImportWizard(QtWidgets.QWizard):
    def __init__(self):
        QtWidgets.QWizard.__init__(self)
        self.setMinimumSize(500, 400)
        self.resize(700, 500)
        self.setPreviewData(None)
        self.selectFilePage = SelectFilePage()
        self.setParametersPage = SetParametersPage()
        self.resultPage = ResultPage()
        self.addPage(self.selectFilePage)
        self.addPage(self.setParametersPage)
        self.addPage(self.resultPage)
        self.setWindowTitle(translate("importwizard", "Import data"))
        self.currentIdChanged.connect(self.onCurrentIdChanged)
    def onCurrentIdChanged(self, id):
        # Hide the 'cancel' button on the last page
        if self.nextId() == -1:
            self.button(QtWidgets.QWizard.CancelButton).hide()
        else:
            self.button(QtWidgets.QWizard.CancelButton).show()
    def open(self, filename):
        if self.isVisible():
            QtWidgets.QMessageBox.information(
                self,
                translate("importwizard", "Import data wizard"),
                translate("importwizard", "The import data wizard is already
open"),
            )
            return
        self.restart()
        self.selectFilePage.txtFilename.setText(filename)
        self.selectFilePage.updatePreview()
        self.show()
    def field(self, name):
        # Allow access to all data via field, some properties are not avaialble
        # as actual controls and therefore we have to handle them ourselves
        if name == "usecols":
            return self.setParametersPage.selectedColumns()
        elif name == "columnnames":
            return self.setParametersPage.columnNames()
        else:
            return QtWidgets.QWizard.field(self, name)
    def setPreviewData(self, data):
        self._previewData = data
    def previewData(self):
        if self._previewData is None:
            raise RuntimeError("Preview data not loaded")
        return self._previewData
if __name__ == "__main__":
    iw = ImportWizard()
    iw.open("test.txt")
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013 Almar Klein
"""
This module defines file system proxies to be used for the file browser.
For now, there is only one: the native file system. But in time,
we may add proxies for ftp, S3, remote computing, etc.
This may seem like an awkward way to use the file system, but (with
small modifications) this approach can probably be used also for
opening/saving files to any file system that we implement here. This
will make Pyzo truly powerful for use in remote computing.
"""
import time
import threading
from queue import Queue, Empty
import os.path as op
from . import QtCore
from .utils import isdir
class Task:
    """Task(**params)
    A task object. Accepts params as keyword arguments.
    When overloading, dont forget to set __slots__.
    Overload and implement the 'process' method to create a task.
    Then use pushTask on a pathProxy object. Use the 'result' method to
    obtain the result (or raise an error).
    """
    __slots__ = ["_params", "_result", "_error"]
    def __init__(self, **params):
        if not params:
            params = None
        self._params = params
        self._result = None
        self._error = None
    def process(self, proxy, **params):
        """process(pathProxy, **params):
        This is the method that represents the task. Overload this to make
        the task do what is intended.
        """
        pass
    def _run(self, proxy):
        """Run the task. Don't overload or use this."""
        try:
            params = self._params or {}
            self._result = self.process(proxy, **params)
        except Exception as err:
            self._error = "Task failed: {}:\n{}".format(self, str(err))
            print(self._error)
    def result(self):
        """Get the result. Raises an error if the task failed."""
        if self._error:
            raise Exception(self._error)
        else:
            return self._result
## Proxy classes for directories and files
class PathProxy(QtCore.QObject):
    """Proxy base class for DirProxy and FileProxy.
    A proxy object is used to get information on a path (folder
    contents, or file modification time), and keep being updated about
    changes in that information.
    One uses an object by connecting to the 'changed' or 'deleted' signal.
    Use setActive(True) to receive updates on these signals. If the proxy
    is no longer needed, use close() to unregister it.
    """
    changed = QtCore.Signal()
    deleted = QtCore.Signal()
    errored = QtCore.Signal(str)  # Or should we pass an error per 'action'?
    taskFinished = QtCore.Signal(Task)
    def __init__(self, fsProxy, path):
        QtCore.QObject.__init__(self)
        self._lock = threading.RLock()
        self._fsProxy = fsProxy
        self._path = path
        self._cancelled = False
        # For tasks
        self._pendingTasks = []
        self._finishedTasks = []
    def __repr__(self):
        return '<{} "{}">'.format(self.__class__.__name__, self._path)
    def path(self):
        """Get the path of this proxy."""
        return self._path
    def track(self):
        """Start tracking this proxy object in the idle time of the
        FSProxy thread.
        """
        self._fsProxy._track(self)
    def push(self):
        """Process this proxy object asap; the object is put in the queue
        of the FSProxy, so it is updated as fast as possible.
        """
        self._cancelled = False
        self._fsProxy._push(self)
    def cancel(self):
        """Stop tracking this proxy object. Cancel processing if this
        object was in the queue.
        """
        self._fsProxy._unTrack(self)
        self._cancelled = True
    def pushTask(self, task):
        """pushTask(task)
        Give a task to the proxy to be executed in the FSProxy
        thread. The taskFinished signal will be emitted with the given
        task when it is done.
        """
        shouldPush = False
        with self._lock:
            if not self._pendingTasks:
                shouldPush = True
            self._pendingTasks.append(task)
        if shouldPush:
            self.push()
    def _processTasks(self):
        # Get pending tasks
        with self._lock:
            pendingTasks = self._pendingTasks
            self._pendingTasks = []
        # Process pending tasks
        finishedTasks = []
        for task in pendingTasks:
            task._run(self)
            finishedTasks.append(task)
        # Emit signal if there are finished tasks
        for task in finishedTasks:
            self.taskFinished.emit(task)
class DirProxy(PathProxy):
    """Proxy object for a directory. Obtain an instance of this class
    using filesystemProx.dir()
    """
    def __init__(self, *args):
        PathProxy.__init__(self, *args)
        self._dirs = set()
        self._files = set()
    def dirs(self):
        with self._lock:
            return set(self._dirs)
    def files(self):
        with self._lock:
            return set(self._files)
    def _process(self, forceUpdate=False):
        # Get info
        dirs = self._fsProxy.listDirs(self._path)
        files = self._fsProxy.listFiles(self._path)
        # Is it deleted?
        if dirs is None or files is None:
            self.deleted.emit()
            return
        # All seems ok. Update if necessary
        dirs, files = set(dirs), set(files)
        if (dirs != self._dirs) or (files != self._files):
            with self._lock:
                self._dirs, self._files = dirs, files
            self.changed.emit()
        elif forceUpdate:
            self.changed.emit()
class FileProxy(PathProxy):
    """Proxy object for a file. Obtain an instance of this class
    using filesystemProx.dir()
    """
    def __init__(self, *args):
        PathProxy.__init__(self, *args)
        self._modified = 0
    def modified(self):
        with self._lock:
            return self._modified
    def _process(self, forceUpdate=False):
        # Get info
        modified = self._fsProxy.modified(self._path)
        # Is it deleted?
        if modified is None:
            self.deleted.emit()
            return
        # All seems ok. Update if necessary
        if modified != self._modified:
            with self._lock:
                self._modified = modified
            self.changed.emit()
        elif forceUpdate:
            self.changed.emit()
    def read(self):
        pass  # ?
    def save(self):
        pass  # ?
## Proxy classes for the file system
class BaseFSProxy(threading.Thread):
    """Abstract base class for file system proxies.
    The file system proxy defines an interface that subclasses can implement
    to "become" a usable file system proxy.
    This class implements the polling of information for the DirProxy
    and FileProxy objects, and keeping them up-to-date. For this purpose
    it keeps a set of PathProxy instances that are polled when idle.
    There is also a queue for items that need processing asap. This is
    where objects are put in when they are activated.
    This class has methods to use the file system (list files and
    directories, etc.). These can be used directly, but may be slow.
    Therefor it is recommended to use the FileProxy and DirProxy objects
    instead.
    """
    # Define how often the registered dirs and files are checked
    IDLE_TIMEOUT = 0.1
    # For testing to induce extra delay. Should normally be close to zero,
    # but not exactly zero!
    IDLE_DELAY = 0.01
    QUEUE_DELAY = 0.01  # 0.5
    def __init__(self):
        threading.Thread.__init__(self)
        self.setDaemon(True)
        #
        self._interrupt = False
        self._exit = False
        #
        self._lock = threading.RLock()
        self._q = Queue()
        self._pathProxies = set()
        #
        self.start()
    def _track(self, pathProxy):
        # todo: use weak references
        with self._lock:
            self._pathProxies.add(pathProxy)
    def _unTrack(self, pathProxy):
        with self._lock:
            self._pathProxies.discard(pathProxy)
    def _push(self, pathProxy):
        # todo: use weak ref here too?
        self._q.put(pathProxy)
        self._interrupt = True
    def stop(self, *, timeout=1.0):
        with self._lock:
            self._exit = True
            self._interrupt = True
            self._pathProxies.clear()
        self.join(timeout)
    def dir(self, path):
        """Convenience function to create a new DirProxy object."""
        return DirProxy(self, path)
    def file(self, path):
        """Convenience function to create a new FileProxy object."""
        return FileProxy(self, path)
    def run(self):
        try:
            try:
                self._run()
            except Exception as err:
                if Empty is None or self._lock is None:
                    pass  # Shutting down ...
                else:
                    print("Exception in proxy thread: " + str(err))
        except Exception:
            pass  # Interpreter is shutting down
    def _run(self):
        last_sleep = time.time()
        while True:
            # Check and reset
            self._interrupt = False
            if self._exit:
                return
            # Sleep
            now = time.time()
            if now - last_sleep > 0.1:
                last_sleep = now
                time.sleep(0.05)
            try:
                # Process items from the queue
                item = self._q.get(True, self.IDLE_TIMEOUT)
                if item is not None and not item._cancelled:
                    self._processItem(item, True)
            except Empty:
                # Queue empty, check items periodically
                self._idle()
    def _idle(self):
        # Make a copy of the set if item
        with self._lock:
            items = set(self._pathProxies)
        # Process them
        for item in items:
            if self._interrupt:
                return
            self._processItem(item)
    def _processItem(self, pathProxy, forceUpdate=False):
        # Slow down a bit
        if forceUpdate:
            time.sleep(self.QUEUE_DELAY)
        else:
            time.sleep(self.IDLE_DELAY)
        # Process
        try:
            pathProxy._process(forceUpdate)
        except Exception as err:
            pathProxy.errored.emit(str(err))
        # Process tasks
        pathProxy._processTasks()
    # To overload ...
    def listDirs(self, path):
        raise NotImplementedError()  # Should rerurn None if it does not exist
    def listFiles(self, path):
        raise NotImplementedError()  # Should rerurn None if it does not exist
    def modified(self, path):
        raise NotImplementedError()  # Should rerurn None if it does not exist
    def fileSize(self, path):
        raise NotImplementedError()  # Should rerurn None if it does not exist
    def read(self, path):
        raise NotImplementedError()  # Should rerurn None if it does not exist
    def write(self, path, bb):
        raise NotImplementedError()
    def rename(self, path):
        raise NotImplementedError()
    def remove(self, path):
        raise NotImplementedError()
    def createDir(self, path):
        raise NotImplementedError()
import os
class NativeFSProxy(BaseFSProxy):
    """File system proxy for the native file system."""
    def listDirs(self, path):
        if isdir(path):
            pp = [op.join(path, p) for p in os.listdir(path)]
            return [str(p) for p in pp if isdir(p)]
    def listFiles(self, path):
        if isdir(path):
            pp = [op.join(path, p) for p in os.listdir(path)]
            return [str(p) for p in pp if op.isfile(p)]
    def modified(self, path):
        if op.isfile(path):
            return op.getmtime(path)
    def fileSize(self, path):
        if op.isfile(path):
            return op.getsize(path)
    def read(self, path):
        if op.isfile(path):
            return open(path, "rb").read()
    def write(self, path, bb):
        with open(path, "wb") as f:
            f.write(bb)
    def rename(self, path1, path2):
        os.rename(path1, path2)
    def remove(self, path):
        if op.isfile(path):
            os.remove(path)
        elif isdir(path):
            os.rmdir(path)
    def createDir(self, path):
        if not isdir(path):
            os.makedirs(path)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013 Almar Klein
"""
Define tasks that can be executed by the file browser.
These inherit from proxies.Task and implement that specific interface.
"""
import re
from . import proxies
class SearchTask(proxies.Task):
    __slots__ = []
    def process(self, proxy, pattern=None, matchCase=False, regExp=False,
**rest):
        # Quick test
        if not pattern:
            return
        # Get text
        text = self._getText(proxy)
        if not text:
            return
        # Get search text. Deal with case sensitivity
        searchText = text
        if not matchCase:
            searchText = searchText.lower()
            pattern = pattern.lower()
        # Search indices
        if regExp:
            indices = self._getIndicesRegExp(searchText, pattern)
        else:
            indices = self._getIndicesNormal1(searchText, pattern)
        # Return as lines
        if indices:
            return self._indicesToLines(text, indices)
        else:
            return []
    def _getText(self, proxy):
        # Init
        path = proxy.path()
        fsProxy = proxy._fsProxy
        # Get file size
        try:
            size = fsProxy.fileSize(path)
        except NotImplementedError:
            pass
        size = size or 0
        # Search all Python files. Other files need be < xx bytes
        if path.lower().endswith(".py") or size < 100 * 1024:
            pass
        else:
            return None
        # Get text
        bb = fsProxy.read(path)
        if bb is None:
            return
        try:
            return bb.decode("utf-8")
        except UnicodeDecodeError:
            # todo: right now we only do utf-8
            return None
    def _getIndicesRegExp(self, text, pattern):
        indices = []
        for match in re.finditer(pattern, text, re.MULTILINE | re.UNICODE):
            indices.append(match.start())
        return indices
    def _getIndicesNormal1(self, text, pattern):
        indices = []
        i = -1
        while True:
            i = text.find(pattern, i + 1)
            if i >= 0:
                indices.append(i)
            else:
                break
        return indices
    def _getIndicesNormal2(self, text, pattern):
        indices = []
        i = 0
        for line in text.splitlines(True):
            i2 = line.find(pattern)
            if i2 >= 0:
                indices.append(i + i2)
            i += len(line)
        return indices
    def _indicesToLines(self, text, indices):
        # Determine line endings
        LE = self._determineLineEnding(text)
        # Obtain line and line numbers
        lines = []
        for i in indices:
            # Get linenr and index of the line
            linenr = text.count(LE, 0, i) + 1
            i1 = text.rfind(LE, 0, i)
            i2 = text.find(LE, i)
            # Get line and strip
            if i1 < 0:
                i1 = 0
            line = text[i1:i2].strip()[:80]
            # Store
            lines.append((linenr, repr(line)))
        # Set result
        return lines
    def _determineLineEnding(self, text):
        """function to determine quickly whether LF or CR is used
        as line endings. Windows endings (CRLF) result in LF
        (you can split lines with either char).
        """
        i = 0
        LE = "\n"
        while i < len(text):
            i += 128
            LF = text.count("\n", 0, i)
            CR = text.count("\r", 0, i)
            if LF or CR:
                if CR > LF:
                    LE = "\r"
                break
        return LE
class PeekTask(proxies.Task):
    """To peek the high level structure of a task."""
    __slots__ = []
    stringStart = re.compile("(\"\"\"|'''|\"|')|#")
    endProgs = {
        "'": re.compile(r"(^|[^\\])(\\\\)*'"),
        '"': re.compile(r'(^|[^\\])(\\\\)*"'),
        "'''": re.compile(r"(^|[^\\])(\\\\)*'''"),
        '"""': re.compile(r'(^|[^\\])(\\\\)*"""'),
    }
    definition = re.compile(r"^(def|class)\s*(\w*)")
    def process(self, proxy):
        path = proxy.path()
        fsProxy = proxy._fsProxy
        # Search only Python files
        if not path.lower().endswith(".py"):
            return None
        # Get text
        bb = fsProxy.read(path)
        if bb is None:
            return
        try:
            text = bb.decode("utf-8")
            del bb
        except UnicodeDecodeError:
            # todo: right now we only do utf-8
            return
        # Parse
        return list(self._parseLines(text.splitlines()))
    def _parseLines(self, lines):
        stringEndProg = None
        linenr = 0
        for line in lines:
            linenr += 1
            # If we are in a triple-quoted multi-line string, find the end
            if stringEndProg is None:
                pos = 0
            else:
                endMatch = stringEndProg.search(line)
                if endMatch is None:
                    continue
                else:
                    pos = endMatch.end()
                    stringEndProg = None
            # Now process all tokens
            while True:
                match = self.stringStart.search(line, pos)
                if (
                    pos == 0
                ):  # If we are at the start of the line, see if we have a top-
level class or method definition
                    end = len(line) if match is None else match.start()
                    definitionMatch = self.definition.search(line[:end])
                    if definitionMatch is not None:
                        if definitionMatch.group(1) == "def":
                            yield (linenr, "def " + definitionMatch.group(2))
                        else:
                            yield (linenr, "class " + definitionMatch.group(2))
                if match is None:
                    break  # Go to next line
                if match.group() == "#":
                    # comment
                    # yield 'C:'
                    break  # Go to next line
                else:
                    endMatch =
self.endProgs[match.group()].search(line[match.end() :])
                    if endMatch is None:
                        if len(match.group()) == 3 or line.endswith("\\"):
                            # Multi-line string
                            stringEndProg = self.endProgs[match.group()]
                            break
                        else:  # incorrect end of single-quoted string
                            break
                    # yield 'S:' + (match.group() +
line[match.end():][:endMatch.end()])
                    pos = match.end() + endMatch.end()
class DocstringTask(proxies.Task):
    __slots__ = []
    def process(self, proxy):
        path = proxy.path()
        fsProxy = proxy._fsProxy
        # Search only Python files
        if not path.lower().endswith(".py"):
            return None
        # Get text
        bb = fsProxy.read(path)
        if bb is None:
            return
        try:
            text = bb.decode("utf-8")
            del bb
        except UnicodeDecodeError:
            # todo: right now we only do utf-8
            return
        # Find docstring
        lines = []
        delim = None  # Not started, in progress, done
        count = 0
        for line in text.splitlines():
            count += 1
            if count > 200:
                break
            # Try to find a start
            if not delim:
                if line.startswith('"""'):
                    delim = '"""'
                    line = line.lstrip('"')
                elif line.startswith("'''"):
                    delim = "'''"
                    line = line.lstrip("'")
            # Try to find an end (may be on the same line as the start)
            if delim and delim in line:
                line = line.split(delim, 1)[0]
                count = 999999999  # Stop; we found the end
            # Add this line
            if delim:
                lines.append(line)
        # Limit number of lines
        if len(lines) > 16:
            lines = lines[:16] + ["..."]
        # Make text and strip
        doc = "\n".join(lines)
        doc = doc.strip()
        return doc
class RenameTask(proxies.Task):
    __slots__ = []
    def process(self, proxy, newpath=None, removeold=False):
        path = proxy.path()
        fsProxy = proxy._fsProxy
        if not newpath:
            return
        if removeold:
            # Works for files and dirs
            fsProxy.rename(path, newpath)
            # The fsProxy will detect that this file is now deleted
        else:
            # Work only for files: duplicate
            # Read bytes
            bb = fsProxy.read(path)
            if bb is None:
                return
            # write back with new name
            fsProxy.write(newpath, bb)
class CreateTask(proxies.Task):
    __slots__ = []
    def process(self, proxy, newpath=None, file=True):
        proxy.path()
        fsProxy = proxy._fsProxy
        if not newpath:
            return
        if file:
            fsProxy.write(newpath, b"")
        else:
            fsProxy.createDir(newpath)
class RemoveTask(proxies.Task):
    __slots__ = []
    def process(self, proxy):
        path = proxy.path()
        fsProxy = proxy._fsProxy
        # Remove
        fsProxy.remove(path)
        # The fsProxy will detect that this file is now deleted
# -*- coding: utf-8 -*-
# Copyright (C) 2013 Almar Klein
"""
Defines the tree widget to display the contents of a selected directory.
"""
import os
import sys
import subprocess
import fnmatch
import os.path as op
import pyzo
from pyzo import translate
from . import QtCore, QtGui, QtWidgets
from . import tasks
from .utils import hasHiddenAttribute, getMounts, cleanpath, isdir, ext
# How to name the list of drives/mounts (i.e. 'my computer')
MOUNTS = "drives"
# Create icon provider
iconprovider = QtWidgets.QFileIconProvider()
def addIconOverlays(icon, *overlays, offset=(8, 0), overlay_offset=(0, 0)):
    """Create an overlay for an icon."""
    # Create painter and pixmap
    pm0 = QtGui.QPixmap(16 + offset[0], 16)  #
icon.pixmap(16+offset[0],16+offset[1])
    pm0.fill(QtGui.QColor(0, 0, 0, 0))
    painter = QtGui.QPainter()
    painter.begin(pm0)
    # Draw original icon
    painter.drawPixmap(offset[0], offset[1], icon.pixmap(16, 16))
    # Draw overlays
    for overlay in overlays:
        pm1 = overlay.pixmap(16, 16)
        painter.drawPixmap(overlay_offset[0], overlay_offset[1], pm1)
    # Finish
    painter.end()
    # Done (return resulting icon)
    return QtGui.QIcon(pm0)
def _filterFileByName(basename, filters):
    # Init default; return True if there are no filters
    default = True
    for filter in filters:
        # Process filters in order
        if filter.startswith("!"):
            # If the filename matches a filter starting with !, hide it
            if fnmatch.fnmatch(basename, filter[1:]):
                return False
            default = True
        else:
            # If the file name matches a filter not starting with!, show it
            if fnmatch.fnmatch(basename, filter):
                return True
            default = False
    return default
def createMounts(browser, tree):
    """Create items for all known mount points (i.e. drives on Windows)."""
    fsProxy = browser._fsProxy
    mountPoints = getMounts()
    mountPoints.sort(key=lambda x: x.lower())
    for entry in mountPoints:
        entry = cleanpath(entry)
        DriveItem(tree, fsProxy.dir(entry))
def createItemsFun(browser, parent):
    """Create the tree widget items for a Tree or DirItem."""
    # Get file system proxy and dir proxy for which we shall create items
    fsProxy = browser._fsProxy
    dirProxy = parent._proxy
    # Get meta information from browser
    searchFilter = browser.searchFilter()
    searchFilter = searchFilter if searchFilter["pattern"] else None
    expandedDirs = browser.expandedDirs
    starredDirs = browser.starredDirs
    # Prepare name filter info
    nameFilters = browser.nameFilter().replace(",", " ").split()
    hideHidden = "!hidden" in nameFilters
    nameFilters = [f for f in nameFilters if f not in ("", "!hiddden",
"hidden")]
    # Filter the contents of this folder
    try:
        dirs = []
        for entry in dirProxy.dirs():
            entry = cleanpath(entry)
            if hideHidden:
                if op.basename(entry).startswith("."):
                    continue  # Skip hidden files
                if hasHiddenAttribute(entry):
                    continue  # Skip hidden files on Windows
                if op.basename(entry) == "node_modules":
                    continue
            if op.basename(entry) == "__pycache__":
                continue
            dirs.append(entry)
        files = []
        for entry in dirProxy.files():
            entry = cleanpath(entry)
            if hideHidden and op.basename(entry).startswith("."):
                continue  # Skip hidden files
            if hideHidden and hasHiddenAttribute(entry):
                continue  # Skip hidden files on Windows
            if not _filterFileByName(op.basename(entry), nameFilters):
                continue
            files.append(entry)
    except (OSError, IOError) as err:
        ErrorItem(parent, str(err))
        return
    # Sort dirs (case insensitive)
    dirs.sort(key=filename2sortkey)
    # Sort files (first by type, then by name, logically)
    files.sort(key=filename2sortkey)
    if not searchFilter:
        # Create dirs
        for path in dirs:
            starred = op.normcase(path) in starredDirs
            item = DirItem(parent, fsProxy.dir(path), starred)
            # Set hidden, we can safely expand programmatically when hidden
            item.setHidden(True)
            # Set expanded and visibility
            if op.normcase(path) in expandedDirs:
                item.setExpanded(True)
            item.setHidden(False)
        # Create files
        for path in files:
            item = FileItem(parent, fsProxy.file(path))
    else:
        # If searching, inject everything in the tree
        # And every item is hidden at first
        parent = browser._tree
        if parent.topLevelItemCount():
            searchInfoItem = parent.topLevelItem(0)
        else:
            searchInfoItem = SearchInfoItem(parent)
        # Increase number of found files
        searchInfoItem.increaseTotal(len(files))
        # Create temporary file items
        for path in files:
            item = TemporaryFileItem(parent, fsProxy.file(path))
            item.search(searchFilter)
        # Create temporary dir items
        if searchFilter["subDirs"]:
            for path in dirs:
                if not os.path.basename(path) in (".git", ".hg"):
                    item = TemporaryDirItem(parent, fsProxy.dir(path))
    # Return number of files added
    return len(dirs) + len(files)
def filename2sortkey(name):
    """Convert a file or dir name to a tuple that can be used to
    logically sort them. Sorting first by extension.
    """
    # Normalize name
    name = os.path.basename(name).lower()
    name, e = os.path.splitext(name)
    # Split the name in logical parts
    try:
        numbers = "0123456789"
        name1 = name.lstrip(numbers)
        name2 = name1.rstrip(numbers)
        n_pre = len(name) - len(name1)
        n_post = len(name1) - len(name2)
        pre = int(name[:n_pre]) if n_pre else 999999999
        post = int(name[-n_post:]) if n_post else -1
        return e, pre, name2, post
    except Exception as err:
        # I cannot see how this could fail, but lets be safe, as it would break
so badly
        print(
            "Warning: could not filename2sortkey(%r), please report:\n%s"
            % (name, str(err))
        )
        return (e, 999999999, name, -1)
class BrowserItem(QtWidgets.QTreeWidgetItem):
    """Abstract item in the tree widget."""
    def __init__(self, parent, pathProxy, *args):
        self._proxy = pathProxy
        QtWidgets.QTreeWidgetItem.__init__(self, parent, [], *args)
        # Set pathname to show, and icon
        strippedParentPath = parent.path().rstrip("/\\")
        if self.path().startswith(strippedParentPath):
            basename = self.path()[len(strippedParentPath) + 1 :]
        else:
            basename = self.path()  #  For mount points
        self.setText(0, basename)
        self.setFileIcon()
        # Setup interface with proxy
        self._proxy.changed.connect(self.onChanged)
        self._proxy.deleted.connect(self.onDeleted)
        self._proxy.errored.connect(self.onErrored)
        self._proxy.taskFinished.connect(self.onTaskFinished)
    def path(self):
        return self._proxy.path()
    def _createDummyItem(self, txt):
        ErrorItem(self, txt)
        # QtWidgets.QTreeWidgetItem(self, [txt])
    def onDestroyed(self):
        self._proxy.cancel()
    def clear(self):
        """Clear method that calls onDestroyed on its children."""
        for i in reversed(range(self.childCount())):
            item = self.child(i)
            if hasattr(item, "onDestroyed"):
                item.onDestroyed()
            self.removeChild(item)
    # To overload ...
    def onChanged(self):
        pass
    def onDeleted(self):
        pass
    def onErrored(self, err):
        self.clear()
        self._createDummyItem("Error: " + err)
    def onTaskFinished(self, task):
        # Getting the result raises exception if an error occured.
        # Which is what we want; so it is visible in the logger shell
        task.result()
class DriveItem(BrowserItem):
    """Tree widget item for directories."""
    def __init__(self, parent, pathProxy):
        BrowserItem.__init__(self, parent, pathProxy)
        # Item is not expandable
    def setFileIcon(self):
        # Use folder icon
        self.setIcon(0, pyzo.icons.drive)
    def onActivated(self):
        self.treeWidget().setPath(self.path())
class DirItem(BrowserItem):
    """Tree widget item for directories."""
    def __init__(self, parent, pathProxy, starred=False):
        self._starred = starred
        BrowserItem.__init__(self, parent, pathProxy)
        # Create dummy item so that the dir is expandable
        self._createDummyItem("Loading contents ...")
    def setFileIcon(self):
        # Use folder icon
        icon = iconprovider.icon(iconprovider.Folder)
        overlays = []
        if self._starred:
            overlays.append(pyzo.icons.bullet_yellow)
        icon = addIconOverlays(icon, *overlays, offset=(8, 0),
overlay_offset=(-4, 0))
        self.setIcon(0, icon)
    def onActivated(self):
        self.treeWidget().setPath(self.path())
    def onExpanded(self):
        # Update list of expanded dirs
        expandedDirs = self.treeWidget().parent().expandedDirs
        p = op.normcase(self.path())  # Normalize case!
        if p not in expandedDirs:
            expandedDirs.append(p)
        # Keep track of changes in our contents
        self._proxy.track()
        self._proxy.push()
    def onCollapsed(self):
        # Update list of expanded dirs
        expandedDirs = self.treeWidget().parent().expandedDirs
        p = op.normcase(self.path())  # Normalize case!
        while p in expandedDirs:
            expandedDirs.remove(p)
        # Stop tracking changes in our contents
        self._proxy.cancel()
        # Clear contents and create a single placeholder item
        self.clear()
        self._createDummyItem("Loading contents ...")
    # No need to implement onDeleted: the parent will get a changed event.
    def onChanged(self):
        """Called when a change in the contents has occured, or when
        we just activated the proxy. Update our items!
        """
        if not self.isExpanded():
            return
        tree = self.treeWidget()
        tree.createItems(self)
class FileItem(BrowserItem):
    """Tree widget item for files."""
    def __init__(self, parent, pathProxy, mode="normal"):
        BrowserItem.__init__(self, parent, pathProxy)
        self._mode = mode
        self._timeSinceLastDocString = 0
        if self._mode == "normal" and self.path().lower().endswith(".py"):
            self._createDummyItem("Loading high level structure ...")
    def setFileIcon(self):
        # Create dummy file in pyzo user dir
        dummy_filename = op.join(
            cleanpath(pyzo.appDataDir), "dummyFiles", "dummy" + ext(self.path())
        )
        # Create file?
        if not op.isfile(dummy_filename):
            if not isdir(op.dirname(dummy_filename)):
                os.makedirs(op.dirname(dummy_filename))
            f = open(dummy_filename, "wb")
            f.close()
        # Use that file
        if sys.platform.startswith("linux") and not
QtCore.__file__.startswith("/usr/"):
            icon = iconprovider.icon(iconprovider.File)
        else:
            icon = iconprovider.icon(QtCore.QFileInfo(dummy_filename))
        icon = addIconOverlays(icon)
        self.setIcon(0, icon)
    def searchContents(self, needle, **kwargs):
        self.setHidden(True)
        self._proxy.setSearch(needle, **kwargs)
    def onActivated(self):
        # todo: someday we should be able to simply pass the proxy object to the
```

```
editors
        # so that we can open files on any file system
        path = self.path()
        if ext(path) not in [".pyc", ".pyo", ".png", ".jpg", ".ico"]:
            # Load file
            pyzo.editors.loadFile(path)
            # Give focus
            pyzo.editors.getCurrentEditor().setFocus()
    def onExpanded(self):
        if self._mode == "normal":
            # Create task to retrieve high level structure
            if self.path().lower().endswith(".py"):
                self._proxy.pushTask(tasks.DocstringTask())
                self._proxy.pushTask(tasks.PeekTask())
    def onCollapsed(self):
        if self._mode == "normal":
            self.clear()
            if self.path().lower().endswith(".py"):
                self._createDummyItem("Loading high level structure ...")
    #     def onClicked(self):
    #         # Limit sending events to prevent flicker when double clicking
    #         if time.time() - self._timeSinceLastDocString < 0.5:
    #             return
    #         self._timeSinceLastDocString = time.time()
    #         # Create task
    #         if self.path().lower().endswith('.py'):
    #             self._proxy.pushTask(tasks.DocstringTask())
    def onChanged(self):
        pass
    def onTaskFinished(self, task):
        if isinstance(task, tasks.DocstringTask):
            result = task.result()
            self.clear()  # Docstring task is done *before* peek task
            if result:
                DocstringItem(self, result)
        #         if isinstance(task, tasks.DocstringTask):
        #             result = task.result()
        #             if result:
        #                 #self.setToolTip(0, result)
        #                 # Show tooltip *now* if mouse is still over this item
        #                 tree = self.treeWidget()
        #                 pos = tree.mapFromGlobal(QtGui.QCursor.pos())
        #                 if tree.itemAt(pos) is self:
```

```
        #                     QtWidgets.QToolTip.showText(QtGui.QCursor.pos(),
result)
        elif isinstance(task, tasks.PeekTask):
            result = task.result()
            # self.clear()  # Cleared when docstring task result is received
            if result:
                for r in result:
                    SubFileItem(self, *r)
            else:
                self._createDummyItem("No classes or functions found.")
        else:
            BrowserItem.onTaskFinished(self, task)
class SubFileItem(QtWidgets.QTreeWidgetItem):
    """Tree widget item for search items."""
    def __init__(self, parent, linenr, text, showlinenr=False):
        QtWidgets.QTreeWidgetItem.__init__(self, parent)
        self._linenr = linenr
        if showlinenr:
            self.setText(0, "Line %i: %s" % (linenr, text))
        else:
            self.setText(0, text)
    def path(self):
        return self.parent().path()
    def onActivated(self):
        path = self.path()
        if ext(path) not in [".pyc", ".pyo", ".png", ".jpg", ".ico"]:
            # Load and get editor
            fileItem = pyzo.editors.loadFile(path)
            editor = fileItem._editor
            # Goto line
            editor.gotoLine(self._linenr)
            # Give focus
            pyzo.editors.getCurrentEditor().setFocus()
class DocstringItem(QtWidgets.QTreeWidgetItem):
    """Tree widget item for docstring placeholder items."""
    def __init__(self, parent, docstring):
        QtWidgets.QTreeWidgetItem.__init__(self, parent)
        self._docstring = docstring
        # Get one-line version of docstring
        shortText = self._docstring.split("\n", 1)[0].strip()
        if len(shortText) < len(self._docstring):
            shortText += "..."
        # Set short version now
        self.setText(0, "doc: " + shortText)
        # Long version is the tooltip
        self.setToolTip(0, docstring)
    def path(self):
        return self.parent().path()
    def onClicked(self):
        tree = self.treeWidget()
        pos = tree.mapFromGlobal(QtGui.QCursor.pos())
        if tree.itemAt(pos) is self:
            QtWidgets.QToolTip.showText(QtGui.QCursor.pos(), self._docstring)
class ErrorItem(QtWidgets.QTreeWidgetItem):
    """Tree widget item for errors and information."""
    def __init__(self, parent, info):
        QtWidgets.QTreeWidgetItem.__init__(self, parent)
        self.setText(0, info)
        self.setFlags(QtCore.Qt.NoItemFlags)
        font = self.font(0)
        font.setItalic(True)
        self.setFont(0, font)
class SearchInfoItem(ErrorItem):
    """Tree widget item that displays info on the search."""
    def __init__(self, parent):
        ErrorItem.__init__(self, parent, "Searching ...")
        self._totalCount = 0
        self._checkCount = 0
        self._hitCount = 0
    def increaseTotal(self, c):
        self._totalCount += c
        self.updateCounts()
    def addFile(self, hit):
        self._checkCount += 1
        if hit:
            self._hitCount += 1
        # Update appearance
        self.updateCounts()
    def updateCounts(self):
        counts = self._checkCount, self._totalCount, self._hitCount
        self.setText(0, "Searched {}/{} files: {} hits".format(*counts))
class TemporaryDirItem:
    """Created when searching. This object posts a requests for its contents
    which are then processed, after which this object disbands itself.
    """
    __slots__ = ["_tree", "_proxy", "__weakref__"]
    def __init__(self, tree, pathProxy):
        self._tree = tree
        self._proxy = pathProxy
        self._proxy.changed.connect(self.onChanged)
        # Process asap, but do not track
        self._proxy.push()
        # Store ourself
        tree._temporaryItems.add(self)
    def clear(self):
        pass  # tree.createItems() calls this ...
    def onChanged(self):
        # Disband
        self._tree._temporaryItems.discard(self)
        # Process contents
        self._tree.createItems(self)
class TemporaryFileItem:
    """Created when searching. This object posts a requests to search
    its contents which are then processed, after which this object
    disbands itself, passin the proxy object to a real FileItem if the
    search had results.
    """
    __slots__ = ["_tree", "_proxy", "__weakref__"]
    def __init__(self, tree, pathProxy):
        self._tree = tree
        self._proxy = pathProxy
        self._proxy.taskFinished.connect(self.onSearchResult)
        # Store ourself
        tree._temporaryItems.add(self)
    def search(self, searchFilter):
        self._proxy.pushTask(tasks.SearchTask(**searchFilter))
    def onSearchResult(self, task):
        # Disband now
        self._tree._temporaryItems.discard(self)
        # Get result. May raise an error
        result = task.result()
        # Process contents
        if result:
            item = FileItem(self._tree, self._proxy, "search")  # Search mode
            for r in result:
                SubFileItem(item, *r, showlinenr=True)
        # Update counter
        searchInfoItem = self._tree.topLevelItem(0)
        if isinstance(searchInfoItem, SearchInfoItem):
            searchInfoItem.addFile(bool(result))
class Tree(QtWidgets.QTreeWidget):
    """Representation of the tree view.
    Instances of this class are responsible for keeping the contents
    up-to-date. The Item classes above are dumb objects.
    """
    dirChanged = QtCore.Signal(str)  # Emitted when user goes into a subdir
    def __init__(self, parent):
        QtWidgets.QTreeWidget.__init__(self, parent)
        # Initialize
        self.setMinimumWidth(150)
        self.setMinimumHeight(150)
        #
        self.setColumnCount(1)
        self.setHeaderHidden(True)
        self.setIconSize(QtCore.QSize(24, 16))
        # Connecy signals
        self.itemExpanded.connect(self.onItemExpanded)
        self.itemCollapsed.connect(self.onItemCollapsed)
        self.itemClicked.connect(self.onItemClicked)
        self.itemActivated.connect(self.onItemActivated)
        # Variables for restoring the view after updating
        self._selectedPath = ""  # To restore a selection after updating
        self._selectedScrolling = 0
        # Set of temporary items
        self._temporaryItems = set()
        # Define context menu
        self.setContextMenuPolicy(QtCore.Qt.CustomContextMenu)
        self.customContextMenuRequested.connect(self.contextMenuTriggered)
        # Initialize proxy (this is where the path is stored)
        self._proxy = None
    def path(self):
        """Get the current path shown by the treeview."""
        return self._proxy.path()
    def setPath(self, path):
        """Set the current path shown by the treeview."""
        # Close old proxy
        if self._proxy is not None:
            self._proxy.cancel()
            self._proxy.changed.disconnect(self.onChanged)
            self._proxy.deleted.disconnect(self.onDeleted)
            self._proxy.errored.disconnect(self.onErrored)
            self.destroyed.disconnect(self._proxy.cancel)
        # Create new proxy
        if True:
            self._proxy = self.parent()._fsProxy.dir(path)
            self._proxy.changed.connect(self.onChanged)
            self._proxy.deleted.connect(self.onDeleted)
            self._proxy.errored.connect(self.onErrored)
            self.destroyed.connect(self._proxy.cancel)
        # Activate the proxy, we'll get a call at onChanged() asap.
        if path.lower() == MOUNTS.lower():
            self.clear()
            createMounts(self.parent(), self)
        else:
            self._proxy.track()
            self._proxy.push()
        # Store dir in config
        self.parent().config.path = path
        # Signal that the dir has changed
        # Note that our contents may not be visible yet.
        self.dirChanged.emit(self.path())
    def setPathUp(self):
        """Go one directory up."""
        newPath = op.dirname(self.path())
        if op.normcase(newPath) == op.normcase(self.path()):
            self.setPath(cleanpath(MOUNTS))
        else:
            self.setPath(newPath)
    def clear(self):
        """Overload the clear method to remove the items in a nice
        way, alowing the pathProxy instance to be closed correctly.
        """
        # Clear temporary (invisible) items
        for item in self._temporaryItems:
            item._proxy.cancel()
        self._temporaryItems.clear()
        # Clear visible items
        for i in reversed(range(self.topLevelItemCount())):
            item = self.topLevelItem(i)
            if hasattr(item, "clear"):
                item.clear()
            if hasattr(item, "onDestroyed"):
                item.onDestroyed()
        QtWidgets.QTreeWidget.clear(self)
    def mouseDoubleClickEvent(self, event):
        """Bypass expanding an item when double-clicking it.
        Only activate the item.
        """
        item = self.itemAt(event.x(), event.y())
        if item is not None:
            self.onItemActivated(item)
    def onChanged(self):
        """Called when our contents change or when we just changed
directories."""
        self.createItems(self)
    def createItems(self, parent):
        """High level method to create the items of the tree or a DirItem.
        This method will handle the restoring of state etc.
        The actual filtering of entries and creation of tree widget items
        is done in the createItemsFun() function.
        """
        # Store state and clear
        self._storeSelectionState()
        parent.clear()
        # Create sub items
        count = createItemsFun(self.parent(), parent)
        if not count and isinstance(parent, QtWidgets.QTreeWidgetItem):
            ErrorItem(parent, "Empty directory")
        # Restore state
        self._restoreSelectionState()
    def onErrored(self, err="..."):
        self.clear()
        ErrorItem(self, "Error: " + err)
    def onDeleted(self):
        self.setPathUp()
    def onItemExpanded(self, item):
        if hasattr(item, "onExpanded"):
            item.onExpanded()
    def onItemCollapsed(self, item):
        if hasattr(item, "onCollapsed"):
            item.onCollapsed()
    def onItemClicked(self, item):
        if hasattr(item, "onClicked"):
            item.onClicked()
    def onItemActivated(self, item):
        """When an item is "activated", make that the new directory,
        or open that file.
        """
        if hasattr(item, "onActivated"):
            item.onActivated()
    def _storeSelectionState(self):
        # Store selection
        items = self.selectedItems()
        self._selectedPath = items[0].path() if items else ""
        # Store scrolling
        self._selectedScrolling = self.verticalScrollBar().value()
    def _restoreSelectionState(self):
        # First select the first item
        # (otherwise the scrolling wont work for some reason)
        if self.topLevelItemCount():
            self.setCurrentItem(self.topLevelItem(0))
        # Restore selection
        if self._selectedPath:
            items = self.findItems(
                op.basename(self._selectedPath), QtCore.Qt.MatchExactly, 0
            )
            items = [
                item
                for item in items
                if op.normcase(item.path()) == op.normcase(self._selectedPath)
            ]
            if items:
                self.setCurrentItem(items[0])
        # Restore scrolling
        self.verticalScrollBar().setValue(self._selectedScrolling)
        self.verticalScrollBar().setValue(self._selectedScrolling)
    def contextMenuTriggered(self, p):
        """Called when context menu is clicked"""
        # Get item that was clicked on
        item = self.itemAt(p)
        if item is None:
            item = self
        # Create and show menu
        if isinstance(item, (Tree, FileItem, DirItem)):
            menu = PopupMenu(self, item)
            menu.popup(self.mapToGlobal(p + QtCore.QPoint(3, 3)))
class PopupMenu(pyzo.core.menu.Menu):
    def __init__(self, parent, item):
        self._item = item
        pyzo.core.menu.Menu.__init__(self, parent, " ")
    def build(self):
        isplat = sys.platform.startswith
        # The star object
        if isinstance(self._item, DirItem):
            if self._item._starred:
                self.addItem(
                    translate("filebrowser", "Unstar this directory"), None,
self._star
                )
            else:
                self.addItem(
                    translate("filebrowser", "Star this directory"), None,
self._star
                )
            self.addSeparator()
        # The pyzo related functions
        if isinstance(self._item, FileItem):
            self.addItem(translate("filebrowser", "Open"), None,
self._item.onActivated)
            if self._item.path().endswith(".py"):
                self.addItem(
                    translate("filebrowser", "Run as script"), None,
self._runAsScript
                )
            elif self._item.path().endswith(".ipynb"):
                self.addItem(
                    translate("filebrowser", "Run Jupyter notebook"),
                    None,
                    self._runNotebook,
                )
            else:
                self.addItem(
                    translate("filebrowser", "Import data..."), None,
self._importData
                )
            self.addSeparator()
        # Create items for open and copy path
        if isinstance(self._item, (FileItem, DirItem)):
            if isplat("win") or isplat("darwin") or isplat("linux"):
                self.addItem(
                    translate("filebrowser", "Open outside Pyzo"),
                    None,
                    self._openOutsidePyzo,
                )
            if isplat("darwin"):
                self.addItem(
                    translate("filebrowser", "Reveal in Finder"),
                    None,
                    self._showInFinder,
                )
            if True:
                self.addItem(
                    translate("filebrowser", "Copy path"), None, self._copyPath
                )
            self.addSeparator()
        # Create items for file management
        if isinstance(self._item, FileItem):
            self.addItem(translate("filebrowser", "Rename"), None,
self.onRename)
            self.addItem(translate("filebrowser", "Delete"), None,
self.onDelete)
            # self.addItem(translate("filebrowser", "Duplicate"), None,
self.onDuplicate)
        if isinstance(self._item, (Tree, DirItem)):
            self.addItem(
                translate("filebrowser", "Create new file"), None,
self.onCreateFile
            )
            self.addItem(
                translate("filebrowser", "Create new directory"), None,
self.onCreateDir
            )
        if isinstance(self._item, DirItem):
            self.addSeparator()
            self.addItem(translate("filebrowser", "Rename"), None,
self.onRename)
            self.addItem(translate("filebrowser", "Delete"), None,
self.onDelete)
    def _star(self):
        # Prepare
        browser = self.parent().parent()
        path = self._item.path()
        if self._item._starred:
            browser.removeStarredDir(path)
        else:
            browser.addStarredDir(path)
        # Refresh
        self.parent().setPath(self.parent().path())
    def _openOutsidePyzo(self):
        path = self._item.path()
        if sys.platform.startswith("darwin"):
            subprocess.call(("open", path))
        elif sys.platform.startswith("win"):
            if " " in path:  # http://stackoverflow.com/a/72796/2271927
                subprocess.call(("start", "", path), shell=True)
            else:
                subprocess.call(("start", path), shell=True)
        elif sys.platform.startswith("linux"):
            # xdg-open is available on all Freedesktop.org compliant distros
            # http://superuser.com/questions/38984/linux-equivalent-command-for-
open-command-on-mac-windows
            subprocess.call(("xdg-open", path))
    def _showInFinder(self):
        subprocess.call(("open", "-R", self._item.path()))
    def _copyPath(self):
        QtWidgets.qApp.clipboard().setText(self._item.path())
    def _runAsScript(self):
        filename = self._item.path()
        shell = pyzo.shells.getCurrentShell()
        if shell is not None:
            shell.restart(filename)
        else:
            msg = "No shell to run code in. "
            m = QtWidgets.QMessageBox(self)
            m.setWindowTitle(translate("menu dialog", "Could not run"))
            m.setText("Could not run " + filename + ":\n\n" + msg)
            m.setIcon(m.Warning)
            m.exec_()
    def _runNotebook(self):
        filename = self._item.path()
        shell = pyzo.shells.getCurrentShell()
        if shell is not None:
            shell.restart(filename)
        else:
            msg = "No shell to run notebook in. "
            m = QtWidgets.QMessageBox(self)
            m.setWindowTitle(translate("menu dialog", "Could not run notebook"))
            m.setText("Could not run " + filename + ":\n\n" + msg)
            m.setIcon(m.Warning)
            m.exec_()
    def _importData(self):
        browser = self.parent().parent()
        wizard = browser.getImportWizard()
        wizard.open(self._item.path())
    def onDuplicate(self):
        return self._duplicateOrRename(False)
    def onRename(self):
        return self._duplicateOrRename(True)
    def onCreateFile(self):
        self._createDirOrFile(True)
    def onCreateDir(self):
        self._createDirOrFile(False)
    def _createDirOrFile(self, file=True):
        # Get title and label
        if file:
            title = translate("filebrowser", "Create new file")
            label = translate("filebrowser", "Give the new name for the file")
        else:
            title = translate("filebrowser", "Create new directory")
            label = translate("filebrowser", "Give the name for the new 
directory")
        # Ask for new filename
        s = QtWidgets.QInputDialog.getText(
            self.parent(),
            title,
            label + ":\n%s" % self._item.path(),
            QtWidgets.QLineEdit.Normal,
            "new name",
        )
        if isinstance(s, tuple):
            s = s[0] if s[1] else ""
        # Push rename task
        if s:
            newpath = op.join(self._item.path(), s)
            task = tasks.CreateTask(newpath=newpath, file=file)
            self._item._proxy.pushTask(task)
    def _duplicateOrRename(self, rename):
        # Get dirname and filename
        dirname, filename = op.split(self._item.path())
        # Get title and label
        if rename:
            title = translate("filebrowser", "Rename")
            label = translate("filebrowser", "Give the new name for the file")
        else:
            title = translate("filebrowser", "Duplicate")
            label = translate("filebrowser", "Give the name for the new file")
            filename = "Copy of " + filename
        # Ask for new filename
        s = QtWidgets.QInputDialog.getText(
            self.parent(),
            title,
            label + ":\n%s" % self._item.path(),
            QtWidgets.QLineEdit.Normal,
            filename,
        )
        if isinstance(s, tuple):
            s = s[0] if s[1] else ""
        # Push rename task
        if s:
            newpath = op.join(dirname, s)
            task = tasks.RenameTask(newpath=newpath, removeold=rename)
            self._item._proxy.pushTask(task)
    def onDelete(self):
        # Ask for new filename
        b = QtWidgets.QMessageBox.question(
            self.parent(),
            translate("filebrowser", "Delete"),
            translate("filebrowser", "Are you sure that you want to delete")
            + ":\n%s" % self._item.path(),
            QtWidgets.QMessageBox.Yes | QtWidgets.QMessageBox.Cancel,
        )
        # Push delete task
        if b == QtWidgets.QMessageBox.Yes:
            self._item._proxy.pushTask(tasks.RemoveTask())
```

```
import os
import ctypes
import sys
import string
import os.path as op
def cleanpath(p):
    return op.normpath(op.expanduser(op.expandvars(p)))
def isdir(p):
    # Add os.sep, because trailing spaces seem to be ignored on Windows
    return op.isdir(p + op.sep)
def ext(p):
    return os.path.splitext(p)[1]
# todo: also include available remote file systems
def getMounts():
    if sys.platform.startswith("win"):
        return getDrivesWin()
    elif sys.platform.startswith("darwin"):
        return "/"
    elif sys.platform.startswith("linux"):
        return ["/"] + [op.join("/media", e) for e in os.listdir("/media")]
    else:
        return "/"
def getDrivesWin():
    drives = []
    bitmask = ctypes.windll.kernel32.GetLogicalDrives()
    for letter in string.ascii_uppercase:
        if bitmask & 1:
            drives.append(letter)
        bitmask >>= 1
    return [drive + ":\\" for drive in drives]
def hasHiddenAttribute(path):
    """Test (on Windows) whether a file should be hidden."""
    if not sys.platform.startswith("win"):
        return False
    try:
        attrs = ctypes.windll.kernel32.GetFileAttributesW(path)
        assert attrs != -1
        return bool(attrs & 2)
    except (AttributeError, AssertionError):
        return False
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
from pyzo import translate
tool_name = translate("pyzoFileBrowser", "File Browser")
tool_summary = "Browse the files in your projects."
""" File browser tool
A powerfull tool for managing your projects in a lightweight manner.
It has a few file management utilities as well.
Config
======
The config consists of three fields:
  * list expandedDirs, with each element a directory
  * list starredDirs, with each element a dict with fields:
      * str path, the directory that is starred
      * str name, the name of the project (op.basename(path) by default)
      * bool addToPythonpath
  * searchMatchCase, searchRegExp, searchSubDirs
  * nameFilter
"""
# todo: List!
"""
  * see easily which files are opened (so it can be used as a secondary tab bar)
  * make visible the "current file" (if applicable)
  * single click on an file that is open selects it in the editor?
  * context menu items to run scripts
  * Support for multiple browsers.
"""
import os.path as op
import pyzo
from pyzo.util import zon as ssdf
from pyzo.qt import QtCore, QtGui, QtWidgets  # noqa
from .browser import Browser
from .utils import cleanpath, isdir
class PyzoFileBrowser(QtWidgets.QWidget):
    """The main tool widget. An instance of this class contains one or
    more Browser instances. If there are more, they can be selected
    using a tab bar.
    """
    def __init__(self, parent):
        QtWidgets.QWidget.__init__(self, parent)
        # Get config
        toolId = self.__class__.__name__.lower() + "2"  # This is v2 of the file
browser
        if toolId not in pyzo.config.tools:
            pyzo.config.tools[toolId] = ssdf.new()
        self.config = pyzo.config.tools[toolId]
        # Ensure three main attributes in config
        for name in ["expandedDirs", "starredDirs"]:
            if name not in self.config:
                self.config[name] = []
        # Ensure path in config
        if "path" not in self.config or not isdir(self.config.path):
            self.config.path = op.expanduser("~")
        # Check expandedDirs and starredDirs.
        # Make path objects and remove invalid dirs. Also normalize case,
        # should not be necessary, but maybe the config was manually edited.
        expandedDirs, starredDirs = [], []
        for d in self.config.starredDirs:
            if "path" in d and "name" in d and "addToPythonpath" in d:
                if isdir(d.path):
                    d.path = op.normcase(cleanpath(d.path))
                    starredDirs.append(d)
        for p in set([str(p) for p in self.config.expandedDirs]):
            if isdir(p):
                p = op.normcase(cleanpath(p))
                # Add if it is a subdir of a starred dir
                for d in starredDirs:
                    if p.startswith(d.path):
                        expandedDirs.append(p)
                        break
        self.config.expandedDirs, self.config.starredDirs = expandedDirs,
starredDirs
        # Create browser(s).
        self._browsers = []
        for i in [0]:
            self._browsers.append(Browser(self, self.config))
        # Layout
        layout = QtWidgets.QVBoxLayout(self)
        self.setLayout(layout)
        layout.addWidget(self._browsers[0])
        layout.setSpacing(0)
        # set margins
        margin = pyzo.config.view.widgetMargin
        layout.setContentsMargins(margin, margin, margin, margin)
    def path(self):
        """Get the current path shown by the file browser."""
        browser = self._browsers[0]
        return browser._tree.path()
    def setPath(self, path):
        """Set the shown path."""
        browser = self._browsers[0]
        browser._tree.setPath(path)
    def getAddToPythonPath(self):
        """
        Returns the path to be added to the Python path when starting a shell
        If a project is selected, which has the addToPath checkbox selected,
        returns the path of the project. Otherwise, returns None
        """
        # Select browser
        browser = self._browsers[0]
        # Select active project
        d = browser.currentProject()
        if d and d.addToPythonpath:
            return d.path
        return None
    def getDefaultSavePath(self):
        """
        Returns the path to be used as default when saving a new file in pyzo.
        Or None if the no path could be determined
        """
        # Select current browser
        browser = self._browsers[0]
        # Select its path
        path = browser._tree.path()
        # Return
        if op.isabs(path) and isdir(path):
            return path
    def closeEvent(self, event):
        self.cleanUp()
        return QtWidgets.QWidget.closeEvent(self, event)
    def cleanUp(self):
        # Close all browsers so they can clean up the file system proxies
        for browser in self._browsers:
            browser.cleanUp()
```

```
"""
Tools to install miniconda from pyzo and register that env in pyzo's
shell config.
"""
import os
import sys
import stat
import time
import struct
import shutil
import threading
import subprocess
import urllib.request
import pyzo
from pyzo.qt import QtCore, QtWidgets
from pyzo import translate
base_url = "http://repo.continuum.io/miniconda/"
links = {
    "win32": "Miniconda3-latest-Windows-x86.exe",
    "win64": "Miniconda3-latest-Windows-x86_64.exe",
    "osx64": "Miniconda3-latest-MacOSX-x86_64.sh",
    "linux32": "Miniconda3-latest-Linux-x86.sh",
    "linux64": "Miniconda3-latest-Linux-x86_64.sh",
    "arm": "Miniconda3-latest-Linux-armv7l.sh",  # raspberry pi
}
# Get where we want to put miniconda installer
miniconda_path = os.path.join(pyzo.appDataDir, "miniconda")
miniconda_path += ".exe" if sys.platform.startswith("win") else ".sh"
# Get default dir where we want the env
# default_conda_dir = os.path.join(pyzo.appDataDir, 'conda_root')
default_conda_dir = (
    "C:\\miniconda3"
    if sys.platform.startswith("win")
    else os.path.expanduser("~/miniconda3")
)
def check_for_conda_env(parent=None):
    """Check if it is reasonable to ask to install a conda env. If
    users says yes, do it. If user says no, don't, and remember.
    """
    # Interested previously?
    if getattr(pyzo.config.state, "did_not_want_conda_env", False):
        print("User has previously indicated to have no interest in a conda
env")
```

```
        return
    # Needed?
    if pyzo.config.shellConfigs2:
        exe = pyzo.config.shellConfigs2[0]["exe"]
        r = ""
        try:
            r = subprocess.check_output(
                [exe, "-m", "conda", "info"],
shell=sys.platform.startswith("win")
            )
            r = r.decode()
        except Exception:
            pass  # no Python or no conda
        if r and "is foreign system : False" in r:
            print("First shell config looks like a conda env.")
            return
    # Ask if interested now?
    d = AskToInstallConda(parent)
    d.exec_()
    if not d.result():
        pyzo.config.state.did_not_want_conda_env = True  # Mark for next time
        return
    # Launch installer
    d = Installer(parent)
    d.exec_()
class AskToInstallConda(QtWidgets.QDialog):
    def __init__(self, parent=None):
        QtWidgets.QDialog.__init__(self, parent)
        self.setWindowTitle("Install a conda env?")
        self.setModal(True)
        text = "Pyzo is only an editor. To execute code, you need a Python
environment.\n\n"
        text += "Do you want Pyzo to install a Python environment
(miniconda)?\n"
        text += "If not, you must arrange for a Python interpreter yourself"
        if not sys.platform.startswith("win"):
            text += " or use the system Python"
        text += "."
        text += "\n(You can always launch the installer from the shell menu.)"
        self._label = QtWidgets.QLabel(text, self)
        self._no = QtWidgets.QPushButton("No thanks (dont ask again)")
        self._yes = QtWidgets.QPushButton("Yes, please install Python!")
        self._no.clicked.connect(self.reject)
        self._yes.clicked.connect(self.accept)
        vbox = QtWidgets.QVBoxLayout(self)
        hbox = QtWidgets.QHBoxLayout()
        self.setLayout(vbox)
        vbox.addWidget(self._label, 1)
        vbox.addLayout(hbox, 0)
        hbox.addWidget(self._no, 2)
        hbox.addWidget(self._yes, 2)
        self._yes.setDefault(1)
class Installer(QtWidgets.QDialog):
    lineFromStdOut = QtCore.Signal(str)
    def __init__(self, parent=None):
        QtWidgets.QDialog.__init__(self, parent)
        self.setWindowTitle("Install miniconda")
        self.setModal(True)
        self.resize(500, 500)
        text = translate(
            "bootstrapconda",
            "This will download and install miniconda on your computer.",
        )
        self._label = QtWidgets.QLabel(text, self)
        self._scipystack = QtWidgets.QCheckBox(
            translate("bootstrapconda", "Also install scientific packages"),
self
        )
        self._scipystack.setChecked(True)
        self._path = QtWidgets.QLineEdit(default_conda_dir, self)
        self._progress = QtWidgets.QProgressBar(self)
        self._outputLine = QtWidgets.QLabel(self)
        self._output = QtWidgets.QPlainTextEdit(self)
        self._output.setReadOnly(True)
        self._button = QtWidgets.QPushButton("Install", self)
        self._outputLine.setSizePolicy(
            QtWidgets.QSizePolicy.Ignored, QtWidgets.QSizePolicy.Fixed
        )
        vbox = QtWidgets.QVBoxLayout(self)
        self.setLayout(vbox)
        vbox.addWidget(self._label, 0)
        vbox.addWidget(self._path, 0)
        vbox.addWidget(self._scipystack, 0)
        vbox.addWidget(self._progress, 0)
        vbox.addWidget(self._outputLine, 0)
        vbox.addWidget(self._output, 1)
        vbox.addWidget(self._button, 0)
        self._button.clicked.connect(self.go)
        self.addOutput(translate("bootstrapconda", "Waiting to start 
installation.\n"))
        self._progress.setVisible(False)
        self.lineFromStdOut.connect(self.setStatus)
    def setStatus(self, line):
        self._outputLine.setText(line)
    def addOutput(self, text):
        # self._output.setPlainText(self._output.toPlainText() + '\n' + text)
        cursor = self._output.textCursor()
        cursor.movePosition(cursor.End, cursor.MoveAnchor)
        cursor.insertText(text)
        cursor.movePosition(cursor.End, cursor.MoveAnchor)
        self._output.setTextCursor(cursor)
        self._output.ensureCursorVisible()
    def addStatus(self, line):
        self.addOutput("\n" + line)
        self.setStatus(line)
    def go(self):
        # Check if we can install
        try:
            self._conda_dir = self._path.text()
            if not os.path.isabs(self._conda_dir):
                raise ValueError("Given installation path must be absolute.")
            if os.path.exists(self._conda_dir):
                raise ValueError("The given installation path already exists.")
        except Exception as err:
            self.addOutput("\nCould not install:\n" + str(err))
            return
        ok = False
        try:
            # Disable user input, get ready for installation
            self._progress.setVisible(True)
            self._button.clicked.disconnect()
            self._button.setEnabled(False)
            self._scipystack.setEnabled(False)
            self._path.setEnabled(False)
            if not os.path.exists(self._conda_dir):
                self.addStatus("Downloading installer ... ")
                self._progress.setMaximum(100)
                self.download()
                self.addStatus("Done downloading installer.")
                self.make_done()
                self.addStatus("Installing (this can take a minute) ... ")
                self._progress.setMaximum(0)
                ret = self.install()
                self.addStatus(("Failed" if ret else "Done") + " installing.")
                self.make_done()
            self.post_install()
            if self._scipystack.isChecked():
                self.addStatus("Installing scientific packages ... ")
                self._progress.setMaximum(0)
                ret = self.install_scipy()
                self.addStatus("Done installing scientific packages")
                self.make_done()
            self.addStatus("Verifying ... ")
            self._progress.setMaximum(100)
            ret = self.verify()
            if ret:
                self.addOutput("Error\n" + ret)
                self.addStatus("Verification Failed!")
            else:
                self.addOutput("Done verifying")
                self.addStatus("Ready to go!")
                self.make_done()
                ok = True
        except Exception as err:
            self.addStatus("Installation failed ...")
            self.addOutput("\n\nException!\n" + str(err))
        if not ok:
            self.addOutput("\n\nWe recommend installing miniconda or anaconda, 
")
            self.addOutput("and making Pyzo aware if it via the shell 
configuration.")
        else:
            self.addOutput(
                '\n\nYou can install additional packages by running "conda 
install" in the shell.'
            )
        # Wrap up, allow user to exit
        self._progress.hide()
        self._button.setEnabled(True)
        self._button.setText("Close")
        self._button.clicked.connect(self.close)
    def make_done(self):
        self._progress.setMaximum(100)
        self._progress.setValue(100)
        etime = time.time() + 0.2
        while time.time() < etime:
            time.sleep(0.01)
            QtWidgets.qApp.processEvents()
    def download(self):
        # Installer already downloaded?
        if os.path.isfile(miniconda_path):
            self.addOutput("Already downloaded.")
            return  # os.remove(miniconda_path)
        # Get url key
        key = ""
        if sys.platform.startswith("win"):
            key = "win"
        elif sys.platform.startswith("darwin"):
            key = "osx"
        elif sys.platform.startswith("linux"):
            key = "linux"
        key += "64" if is_64bit() else "32"
        # Get url
        if key not in links:
            raise RuntimeError("Cannot download miniconda for this platform.")
        url = base_url + links[key]
        _fetch_file(url, miniconda_path, self._progress)
    def install(self):
        dest = self._conda_dir
        # Clear dir
        assert not os.path.isdir(dest), "Miniconda dir already exists"
        assert " " not in dest, "miniconda dest path must not contain spaces"
        if sys.platform.startswith("win"):
            return self._run_process([miniconda_path, "/S", "/D=%s" % dest])
        else:
            os.chmod(miniconda_path, os.stat(miniconda_path).st_mode |
stat.S_IEXEC)
            return self._run_process([miniconda_path, "-b", "-p", dest])
    def post_install(self):
        exe = py_exe(self._conda_dir)
        # Add Pyzo channel
        cmd = [exe, "-m", "conda", "config", "--system", "--add", "channels",
"pyzo"]
        subprocess.check_call(cmd, shell=sys.platform.startswith("win"))
        self.addStatus("Added Pyzo channel to conda env")
        # Add to pyzo shell config
        if pyzo.config.shellConfigs2 and pyzo.config.shellConfigs2[0]["exe"] ==
exe:
            pass
        else:
            s = pyzo.ssdf.new()
            s.name = "Py3-conda"
            s.exe = exe
            s.gui = "PyQt4"
            pyzo.config.shellConfigs2.insert(0, s)
            pyzo.saveConfig()
            self.addStatus("Prepended new env to Pyzo shell configs.")
    def install_scipy(self):
        packages = [
            "numpy",
            "scipy",
            "pandas",
            "matplotlib",
            "sympy",
            #'scikit-image', 'scikit-learn',
            "pyopengl",  # 'visvis', 'imageio',
            "tornado",
            "pyqt",  #'ipython', 'jupyter',
            #'requests', 'pygments','pytest',
        ]
        exe = py_exe(self._conda_dir)
        cmd = [exe, "-m", "conda", "install", "--yes"] + packages
        return self._run_process(cmd)
    def _run_process(self, cmd):
        """Run command in a separate process, catch stdout, show lines
        in the output label. On fail, show all output in output text.
        """
        p = subprocess.Popen(
            cmd,
            stdout=subprocess.PIPE,
            stderr=subprocess.STDOUT,
            shell=sys.platform.startswith("win"),
        )
        catcher = StreamCatcher(p.stdout, self.lineFromStdOut)
        while p.poll() is None:
            time.sleep(0.01)
            QtWidgets.qApp.processEvents()
        catcher.join()
        if p.poll():
            self.addOutput(catcher.output())
        return p.poll()
    def verify(self):
        self._progress.setValue(1)
        if not os.path.isdir(self._conda_dir):
            return "Conda dir not created."
        self._progress.setValue(11)
        exe = py_exe(self._conda_dir)
        if not os.path.isfile(exe):
            return "Conda dir does not have Python exe"
        self._progress.setValue(21)
        try:
            ver = subprocess.check_output([exe, "-c", "import sys;
print(sys.version)"])
        except Exception as err:
            return "Error getting Python version: " + str(err)
        self._progress.setValue(31)
        if ver.decode() < "3.4":
            return "Expected Python version 3.4 or higher"
        self._progress.setValue(41)
        try:
            ver = subprocess.check_output(
                [exe, "-c", "import conda; print(conda.__version__)"]
            )
        except Exception as err:
            return "Error calling Python exe: " + str(err)
        self._progress.setValue(51)
        if ver.decode() < "3.16":
            return "Expected Conda version 3.16 or higher"
        # Smooth toward 100%
        for i in range(self._progress.value(), 100, 5):
            time.sleep(0.05)
            self._progress.setValue(i)
            QtWidgets.qApp.processEvents()
def is_64bit():
    """Get whether the OS is 64 bit. On WIndows yields what it *really*
    is, not what the process is.
    """
    if False:  # sys.platform.startswith('win'):  ARG, causes problems with
subprocess
        if "PROCESSOR_ARCHITEW6432" in os.environ:
            return True
        return os.environ["PROCESSOR_ARCHITECTURE"].endswith("64")
    else:
        return struct.calcsize("P") == 8
def py_exe(dir):
    if sys.platform.startswith("win"):
        return os.path.join(dir, "python.exe")
    else:
        return os.path.join(dir, "bin", "python")
def _chunk_read(response, local_file, chunk_size=1024, initial_size=0,
progress=None):
    """Download a file chunk by chunk and show advancement"""
    # Adapted from NISL:
    # https://github.com/nisl/tutorial/blob/master/nisl/datasets.py
    bytes_so_far = initial_size
    # Returns only amount left to download when resuming, not the size of the
    # entire file
    total_size = int(response.headers["Content-Length"].strip())
    total_size += initial_size
    if progress:
        progress.setMaximum(total_size)
    while True:
        QtWidgets.qApp.processEvents()
        chunk = response.read(chunk_size)
        bytes_so_far += len(chunk)
        if not chunk:
            sys.stderr.write("\n")
            break
        # _chunk_write(chunk, local_file, progress)
        progress.setValue(bytes_so_far)
        local_file.write(chunk)
def _fetch_file(url, file_name, progress=None):
    """Load requested file, downloading it if needed or requested"""
    # Adapted from NISL:
    # https://github.com/nisl/tutorial/blob/master/nisl/datasets.py
    temp_file_name = file_name + ".part"
    local_file = None
    initial_size = 0
    try:
        # Checking file size and displaying it alongside the download url
        response = urllib.request.urlopen(url, timeout=5.0)
        # file_size = int(response.headers['Content-Length'].strip())
        # Downloading data (can be extended to resume if need be)
        local_file = open(temp_file_name, "wb")
        _chunk_read(response, local_file, initial_size=initial_size,
progress=progress)
        # temp file must be closed prior to the move
        if not local_file.closed:
            local_file.close()
        shutil.move(temp_file_name, file_name)
    except Exception as e:
        raise RuntimeError(
            "Error while fetching file %s.\n" "Dataset fetching aborted (%s)" %
(url, e)
        )
    finally:
        if local_file is not None:
            if not local_file.closed:
                local_file.close()
class StreamCatcher(threading.Thread):
    def __init__(self, file, signal):
        self._file = file
        self._signal = signal
        self._lines = []
        self._line = ""
        threading.Thread.__init__(self)
        self.setDaemon(True)  # do not let this thread hold up Python shutdown
        self.start()
    def run(self):
        while True:
            time.sleep(0.0001)
            try:
                part = self._file.read(20)
            except ValueError:  # pragma: no cover
                break
            if not part:
                break
            part = part.decode("utf-8", "ignore")
            # print(part, end='')
            self._line += part.replace("\r", "\n")
            lines = [line for line in self._line.split("\n") if line]
            self._lines.extend(lines[:-1])
            self._line = lines[-1]
            if self._lines:
                self._signal.emit(self._lines[-1])
        self._lines.append(self._line)
        self._signal.emit(self._lines[-1])
    def output(self):
        return "\n".join(self._lines)
if __name__ == "__main__":
    check_for_conda_env()
```

```
# -*- coding: utf-8 -*-
# Copyright (c) 2016, Almar Klein, Rob Reilink
#
# This file is distributed under the terms of the 2-Clause BSD License.
""" Module paths
Get paths to useful directories in a cross platform manner. The functions
in this module are designed to be stand-alone, so that they can easily
be copied and used in code that does not want pyzo as a dependency.
This code was first part of pyzolib, and later moved to pyzo.
"""
# Notes:
# * site.getusersitepackages() returns a dir in roaming userspace on Windows
#   so better avoid that.
# * site.getuserbase() returns appdata_dir('Python', True)
# * See docstring: that's why the functions tend to not re-use each-other
import sys
from pyzo.qt import QtCore
ISWIN = sys.platform.startswith("win")
ISMAC = sys.platform.startswith("darwin")
ISLINUX = sys.platform.startswith("linux")
PY2 = sys.version_info[0] == 2
PY3 = sys.version_info[0] == 3
# From pyzolib/paths.py (https://bitbucket.org/pyzo/pyzolib/src/tip/paths.py)
import sys
def is_frozen():
    """is_frozen()
    Return whether this app is a frozen application (using e.g. cx_freeze).
    """
    return bool(getattr(sys, "frozen", None))
# From pyzolib/paths.py (https://bitbucket.org/pyzo/pyzolib/src/tip/paths.py)
import os, sys, tempfile
def temp_dir(appname=None, nospaces=False):
    """temp_dir(appname=None, nospaces=False)
    Get path to a temporary directory with write access.
    If appname is given, a subdir is appended (and created if necessary).
    If nospaces, will ensure that the path has no spaces.
    """
    # Do it the Python way
    path = tempfile.gettempdir()
    # Try harder if we have to
    if nospaces and " " in path:
        if sys.platform.startswith("win"):
            for path in ["c:\\TEMP", "c:\\TMP"]:
                if os.path.isdir(path):
                    break
            os.makedirs(path, exist_ok=True)
        else:
            for path in [
                "/tmp",
                "/var/tmp",
            ]:  # http://www.tuxfiles.org/linuxhelp/linuxdir.html
                if os.path.isdir(path):
                    break
            else:
                raise RuntimeError("Could not locate temporary directory.")
    # Get path specific for this app
    if appname:
        path = os.path.join(path, appname)
        os.makedirs(path, exist_ok=True)
    # Done
    return path
# From pyzolib/paths.py (https://bitbucket.org/pyzo/pyzolib/src/tip/paths.py)
import os
def user_dir():
    """user_dir()
    Get the path to the user directory. (e.g. "/home/jack", "c:/Users/jack")
    """
    return os.path.expanduser("~")
# From pyzolib/paths.py (https://bitbucket.org/pyzo/pyzolib/src/tip/paths.py)
import os, sys
def appdata_dir(appname=None, roaming=False, macAsLinux=False):
    """appdata_dir(appname=None, roaming=False,  macAsLinux=False)
    Get the path to the application data and config directory, where 
applications are allowed
    to write user specific files (e.g. configurations).
    Applications should write their configurations files in the config folder,
    and other data (e.g. history files) in the data folder.
    For non-user specific data, consider using common_appdata_dir().
    If appname is given, a subdir is appended (and created if necessary).
    If roaming is True, will prefer a roaming directory (Windows Vista/7).
    If macAsLinux is True, will return the Linux-like location on Mac.
    The behaviour of this function changed, it now uses QStandardPaths to 
provide location
    of data folder and config folder, but for retro-compatibility pyzo will use 
the old folder if it exists
    """
    # Define default user directory
    userDir = os.path.expanduser("~")
    # Get system app data dir
    path = None
    if sys.platform.startswith("win"):
        path1, path2 = os.getenv("LOCALAPPDATA"), os.getenv("APPDATA")
        path = (path2 or path1) if roaming else (path1 or path2)
    elif sys.platform.startswith("darwin") and not macAsLinux:
        path = os.path.join(userDir, "Library", "Application Support")
    # On Linux and as fallback
    if not (path and os.path.isdir(path)):
        path = userDir
    # Maybe we should store things local to the executable (in case of a
    # portable distro or a frozen application that wants to be portable)
    prefix = sys.prefix
    if getattr(sys, "frozen", None):  # See application_dir() function
        prefix = os.path.abspath(os.path.dirname(sys.executable))
    for reldir in ("settings", "../settings"):
        localpath = os.path.abspath(os.path.join(prefix, reldir))
        if os.path.isdir(localpath):
            try:
                open(os.path.join(localpath, "test.write"), "wb").close()
                os.remove(os.path.join(localpath, "test.write"))
            except IOError:
                pass  # We cannot write in this directory
            else:
                path = localpath
                break
    data_path, config_path = path, path
    # Get path specific for this app
    if appname:
        if path == userDir:
            appname = "." + appname.lstrip(".")  # Make it a hidden directory
        path = os.path.join(path, appname)
        data_path, config_path = path, path
        if not os.path.isdir(path):
            # Better way to get config/data folder especially on *nix system
(see XDG_CONFIG_HOME standard),
            # but this should work on any os.
            # For retro-compatibility, check if old folder exist, and if not,
use standard path.
            try:
                standard_data_path = QtCore.QStandardPaths.writableLocation(
                    QtCore.QStandardPaths.AppDataLocation
                )
                standard_config_path = QtCore.QStandardPaths.writableLocation(
                    QtCore.QStandardPaths.ConfigLocation
                )
            except AttributeError:
                pass
            else:
                # Check if QStandardPaths succeeded to find the location,
otherwise use old path
                if standard_config_path:
                    config_path = standard_config_path
                if standard_data_path:
                    data_path = standard_data_path
                appname = appname.lstrip(".")
                data_path = os.path.join(data_path, appname)
                config_path = os.path.join(config_path, appname)
            os.makedirs(data_path, exist_ok=True)
            os.makedirs(config_path, exist_ok=True)
    # Done
    return data_path, config_path
# From pyzolib/paths.py (https://bitbucket.org/pyzo/pyzolib/src/tip/paths.py)
import os, sys
def common_appdata_dir(appname=None):
    """common_appdata_dir(appname=None)
    Get the path to the common application directory. Applications are
    allowed to write files here. For user specific data, consider using
    appdata_dir().
    If appname is given, a subdir is appended (and created if necessary).
    """
    # Try to get data_path
    data_path = None
    if sys.platform.startswith("win"):
        data_path = os.getenv("ALLUSERSPROFILE", os.getenv("PROGRAMDATA"))
    elif sys.platform.startswith("darwin"):
        data_path = "/Library/Application Support"
    else:
        # Not sure what to use. Apps are only allowed to write to the home
        # dir and tmp dir, right?
        pass
    # If no success, use appdata_dir() instead
    if not (data_path and os.path.isdir(data_path)):
        data_path = appdata_dir()[0]
    # Get path specific for this app
    if appname:
        data_path = os.path.join(data_path, appname)
        os.makedirs(data_path, exist_ok=True)
    # Done
    return data_path
#  Other approaches that we considered, but which did not work for links,
#  or are less reliable for other reasons are:
#      * sys.executable: does not work for links
#      * sys.prefix: dito
#      * sys.exec_prefix: dito
#      * os.__file__: does not work when frozen
#      * __file__: only accessable from main module namespace, does not work
when frozen
# todo: get this included in Python sys or os module!
# From pyzolib/paths.py (https://bitbucket.org/pyzo/pyzolib/src/tip/paths.py)
import os, sys
def application_dir():
    """application_dir()
    Get the directory in which the current application is located.
    The "application" can be a Python script or a frozen application.
    This function raises a RuntimeError if in interpreter mode.
    """
    if getattr(sys, "frozen", False):
        # When frozen, use sys.executable
        thepath = os.path.dirname(sys.executable)
    else:
        # Test if the current process can be considered an "application"
        if not sys.path or not sys.path[0]:
            raise RuntimeError(
                "Cannot determine app path because sys.path[0] is empty!"
            )
        thepath = sys.path[0]
    # Return absolute version, or symlinks may not work
    return os.path.abspath(thepath)
## Pyzo specific
#
# A Pyzo distribution maintains a file in the appdata dir that lists
# the directory where it is intalled. Pyzo can in principle be installed
# multiple times. In that case the file contains multiple entries.
# This file is checked each time the pyzo executable is run. Therefore
# a user can move the Pyzo directory and simply run the Pyzo executable
# to update the registration.
```

```
def pyzo_dirs(newdir=None, makelast=False):
    """pyzo_dirs(newdir=None,  makelast=False)
    Compatibility function. Like pyzo_dirs2, but returns a list of
    directories and does not allow setting the version.
    """
    return [p[0] for p in pyzo_dirs2(newdir, makelast=makelast)]
# From pyzolib/paths.py (https://bitbucket.org/pyzo/pyzolib/src/tip/paths.py)
import os, sys
def pyzo_dirs2(path=None, version="0", **kwargs):
    """pyzo_dirs2(dir=None, version='0', makelast=False)
    Get the locations of installed Pyzo directories. Returns a list of
    tuples: (dirname, version). In future versions more information may
    be added to the file, so please take larger tuples into account.
    If path is a dir containing a python exe, it is added it to the
    list. If the keyword arg 'makelast' is given and True, will ensure
    that the given path is the last in the list (i.e. the default).
    """
    defaultPyzo = "", "0"  # To fill in values for shorter items
    newPyzo = (str(path), str(version)) if path else None
    # Get application dir
    userDir = os.path.expanduser("~")
    path = None
    if sys.platform.startswith("win"):
        path = os.getenv("LOCALAPPDATA", os.getenv("APPDATA"))
    elif sys.platform.startswith("darwin"):
        path = os.path.join(userDir, "Library", "Application Support")
    # Get application dir for Pyzo
    if path and os.path.isdir(path):
        path = os.path.join(path, "pyzo")
    else:
        path = os.path.join(userDir, ".pyzo")  # On Linux and as fallback
    if not os.path.isdir(path):
        # Better way to get config folder especially on *nix system (see
XDG_CONFIG_HOME standard),
        # but this should work on any os.
        # For retro-compatibility, check if old folder exist and use new path
otherwise.
        standard_path = QtCore.QStandardPaths.writableLocation(
            QtCore.QStandardPaths.ConfigLocation
        )
        if (
            standard_path != ""
        ):  # Check if QStandardPaths succeeded to find the location, otherwise
```

```
use old path
            path = standard_path
        path = os.path.join(path, "pyzo")
        os.makedirs(path, exist_ok=True)
    # Open file and parse
    fname = os.path.join(path, "pyzodirs")
    pyzos, npyzos = [], 0
    if os.path.isfile(fname):
        lines = open(fname, "rb").read().decode("utf-8").split("\n")
        pyzos = [tuple(d.split(":::")) for d in [d.strip() for d in lines] if d]
        npyzos = len(pyzos)
    # Add dir if necessary
    if newPyzo and os.path.isdir(newPyzo[0]):
        if kwargs.get("makelast", False) or newPyzo not in pyzos:
            npyzos = 0  # force save
            pyzos = [p for p in pyzos if p[0] != newPyzo[0]]  # rm based on dir
            pyzos.append(newPyzo)
    # Check validity of all pyzos, write back if necessary, and return
    pythonname = "python" + ".exe" * sys.platform.startswith("win")
    pyzos = [p for p in pyzos if os.path.isfile(os.path.join(p[0], pythonname))]
    if len(pyzos) != npyzos:
        lines = [":::".join(p) for p in pyzos]
        open(fname, "wb").write(("\n".join(lines)).encode("utf-8"))
    return [p + defaultPyzo[len(p) :] for p in pyzos]
## Windows specific
# Maybe for directory of programs, pictures etc.
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" pyzowizard module
Implements a wizard to help new users get familiar with pyzo.
"""
import os
import re
import pyzo
from pyzo.qt import QtCore, QtGui, QtWidgets
from pyzo import translate
from pyzo.util._locale import LANGUAGES, LANGUAGE_SYNONYMS, setLanguage
def retranslate(t):
    """To allow retranslating after selecting the language."""
    if hasattr(t, "original"):
        return translate("wizard", t.original)
    else:
        return t
class PyzoWizard(QtWidgets.QWizard):
    def __init__(self, parent):
        QtWidgets.QWizard.__init__(self, parent)
        # Set some appearance stuff
        self.setMinimumSize(600, 500)
        self.setWindowTitle(translate("wizard", "Getting started with Pyzo"))
        self.setWizardStyle(self.ModernStyle)
        self.setButtonText(self.CancelButton, "Stop")
        # Set logo
        pm = QtGui.QPixmap()
        pm.load(os.path.join(pyzo.pyzoDir, "resources", "appicons",
"pyzologo48.png"))
        self.setPixmap(self.LogoPixmap, pm)
        # Define pages
        klasses = [
            IntroWizardPage,
            TwocomponentsWizardPage,
            EditorWizardPage,
            ShellWizardPage1,
            ShellWizardPage2,
            RuncodeWizardPage1,
            RuncodeWizardPage2,
            ToolsWizardPage1,
            ToolsWizardPage2,
            FinalPage,
        ]
        # Create pages
        self._n = len(klasses)
        for i, klass in enumerate(klasses):
            self.addPage(klass(self, i))
    def show(self, startPage=None):
        """Show the wizard. If startPage is given, open the Wizard at
        that page. startPage can be an integer or a string that matches
        the classname of a page.
        """
        QtWidgets.QWizard.show(self)
        # Check startpage
        if isinstance(startPage, int):
            pass
        elif isinstance(startPage, str):
            for i in range(self._n):
                page = self.page(i)
                if page.__class__.__name__.lower() == startPage.lower():
                    startPage = i
                    break
            else:
                print("Pyzo wizard: Could not find start page: %r" % startPage)
                startPage = None
        elif startPage is not None:
            print("Pyzo wizard: invalid start page: %r" % startPage)
            startPage = None
        # Go to start page
        if startPage is not None:
            for i in range(startPage):
                self.next()
class BasePyzoWizardPage(QtWidgets.QWizardPage):
    _prefix = translate("wizard", "Step")
    _title = "dummy title"
    _descriptions = []
    _image_filename = ""
    def __init__(self, parent, i):
        QtWidgets.QWizardPage.__init__(self, parent)
        self._i = i
        # Create label for description
        self._text_label = QtWidgets.QLabel(self)
        self._text_label.setTextFormat(QtCore.Qt.RichText)
        self._text_label.setWordWrap(True)
        # Create label for image
        self._comicLabel = QtWidgets.QLabel(self)
        pm = QtGui.QPixmap()
        if "logo" in self._image_filename:
            pm.load(
                os.path.join(
                    pyzo.pyzoDir, "resources", "appicons", self._image_filename
                )
            )
        elif self._image_filename:
            pm.load(
                os.path.join(pyzo.pyzoDir, "resources", "images",
self._image_filename)
            )
        self._comicLabel.setPixmap(pm)
        self._comicLabel.setAlignment(QtCore.Qt.AlignHCenter |
QtCore.Qt.AlignVCenter)
        # Layout
        theLayout = QtWidgets.QVBoxLayout(self)
        self.setLayout(theLayout)
        #
        theLayout.addWidget(self._text_label)
        theLayout.addStretch()
        theLayout.addWidget(self._comicLabel)
        theLayout.addStretch()
    def initializePage(self):
        # Get prefix
        i = self._i
        n = self.wizard()._n - 2  # Dont count the first and last page
        prefix = ""
        if i and i <= n:
            prefix = retranslate(self._prefix) + " %i/%i: " % (i, n)
        # Set title
        self.setTitle(prefix + retranslate(self._title))
        # Parse description
        # Two description units are separated with BR tags
        # Emphasis on words is set to italic tags.
        lines = []
        descriptions = [retranslate(d).strip() for d in self._descriptions]
        for description in descriptions:
            for line in description.splitlines():
                line = line.strip()
                line = re.sub(r"\*(.+?)\*", r"<b>\1</b>", line)
                lines.append(line)
            lines.append("<br /><br />")
        lines = lines[:-1]
        # Set description
        self._text_label.setText("\n".join(lines))
class IntroWizardPage(BasePyzoWizardPage):
    _title = translate("wizard", "Welcome to the Interactive Editor for 
Python!")
    _image_filename = "pyzologo128.png"
    _descriptions = [
        translate(
            "wizard",
            """This wizard helps you get familiarized with the workings of 
Pyzo.""",
        ),
        translate(
            "wizard",
            """Pyzo is a cross-platform Python IDE
        focused on *interactivity* and *introspection*, which makes it
        very suitable for scientific computing. Its practical design
        is aimed at *simplicity* and *efficiency*.""",
        ),
    ]
    def __init__(self, parent, i):
        BasePyzoWizardPage.__init__(self, parent, i)
        # Create label and checkbox
        t1 = translate("wizard", "This wizard can be opened using 'Help > Pyzo 
wizard'")
        # t2 = translate('wizard', "Show this wizard on startup")
        self._label_info = QtWidgets.QLabel(t1, self)
        # self._check_show = QtWidgets.QCheckBox(t2, self)
        # self._check_show.stateChanged.connect(self._setNewUser)
        # Create language switcher
        self._langLabel = QtWidgets.QLabel(translate("wizard", "Select 
language"), self)
        #
        self._langBox = QtWidgets.QComboBox(self)
        self._langBox.setEditable(False)
        # Fill
        index, theIndex = -1, -1
        cur = pyzo.config.settings.language
        for lang in sorted(LANGUAGES):
            index += 1
            self._langBox.addItem(lang)
            if lang == LANGUAGE_SYNONYMS.get(cur, cur):
                theIndex = index
        # Set current index
        if theIndex >= 0:
            self._langBox.setCurrentIndex(theIndex)
        # Bind signal
        self._langBox.activated.connect(self.onLanguageChange)
        # Init check state
        # if pyzo.config.state.newUser:
        #    self._check_show.setCheckState(QtCore.Qt.Checked)
        # Create sublayout
        layout = QtWidgets.QHBoxLayout()
        layout.addWidget(self._langLabel, 0)
        layout.addWidget(self._langBox, 0)
        layout.addStretch(2)
        self.layout().addLayout(layout)
        # Add to layout
        self.layout().addSpacing(10)
        self.layout().addWidget(self._label_info)
        # self.layout().addWidget(self._check_show)
    def _setNewUser(self, newUser):
        newUser = bool(newUser)
        self._label_info.setHidden(newUser)
        pyzo.config.state.newUser = newUser
    def onLanguageChange(self):
        languageName = self._langBox.currentText()
        if pyzo.config.settings.language == languageName:
            return
        # Save new language
        pyzo.config.settings.language = languageName
        setLanguage(pyzo.config.settings.language)
        # Notify user
        text = translate(
            "wizard",
            """
        The language has been changed for this wizard.
        Pyzo needs to restart for the change to take effect application-wide.
        """,
        )
        m = QtWidgets.QMessageBox(self)
        m.setWindowTitle(translate("wizard", "Language changed"))
        m.setText(text)
        m.setIcon(m.Information)
        m.exec_()
        # Get props of current wizard
        geo = self.wizard().geometry()
        parent = self.wizard().parent()
        # Close ourself!
        self.wizard().close()
        # Start new one
        w = PyzoWizard(parent)
        w.setGeometry(geo)
        w.show()
class TwocomponentsWizardPage(BasePyzoWizardPage):
    _title = translate("wizard", "Pyzo consists of two main components")
    _image_filename = "pyzo_two_components.png"
    _descriptions = [
        translate("wizard", "You can execute commands directly in the
*shell*,"),
        translate("wizard", "or you can write code in the *editor* and execute
that."),
    ]
class EditorWizardPage(BasePyzoWizardPage):
    _title = translate("wizard", "The editor is where you write your code")
    _image_filename = "pyzo_editor.png"
    _descriptions = [
        translate(
            "wizard",
            """In the *editor*, each open file is represented as a tab. By
        right-clicking on a tab, files can be run, saved, closed, etc.""",
        ),
        translate(
            "wizard",
            """The right mouse button also enables one to make a file the
        *main file* of a project. This file can be recognized by its star
        symbol, and it enables running the file more easily.""",
        ),
    ]
class ShellWizardPage1(BasePyzoWizardPage):
    _title = translate("wizard", "The shell is where your code gets executed")
    _image_filename = "pyzo_shell1.png"
    _descriptions = [
        translate(
            "wizard",
            """When Pyzo starts, a default *shell* is created. You can add more
        shells that run simultaneously, and which may be of different
        Python versions.""",
        ),
        translate(
            "wizard",
            """Shells run in a sub-process, such that when it is busy, Pyzo
        itself stays responsive, allowing you to keep coding and even
        run code in another shell.""",
        ),
    ]
class ShellWizardPage2(BasePyzoWizardPage):
    _title = translate("wizard", "Configuring shells")
    _image_filename = "pyzo_shell2.png"
    _descriptions = [
        translate(
            "wizard",
            """Pyzo can integrate the event loop of five different *GUI
toolkits*,
        thus enabling interactive plotting with e.g. Visvis or Matplotlib.""",
        ),
        translate(
            "wizard",
            """Via 'Shell > Edit shell configurations', you can edit and add
        *shell configurations*. This allows you to for example select the
        initial directory, or use a custom Pythonpath.""",
        ),
    ]
class RuncodeWizardPage1(BasePyzoWizardPage):
    _title = translate("wizard", "Running code")
    _image_filename = "pyzo_run1.png"
    _descriptions = [
        translate(
            "wizard",
            "Pyzo supports several ways to run source code in the editor. (see
the 'Run' menu).",
        ),
        translate(
            "wizard",
            """*Run selection:* if there is no selected text, the current line
        is executed; if the selection is on a single line, the selection
        is evaluated; if the selection spans multiple lines, Pyzo will
        run the the (complete) selected lines.""",
        ),
        translate(
            "wizard",
            "*Run cell:* a cell is everything between two lines starting with
'##'.",
        ),
        translate("wizard", "*Run file:* run all the code in the current
file."),
        translate(
            "wizard",
            "*Run project main file:* run the code in the current project's main
file.",
        ),
    ]
class RuncodeWizardPage2(BasePyzoWizardPage):
    _title = translate("wizard", "Interactive mode vs running as script")
    _image_filename = ""
    _descriptions = [
        translate(
            "wizard",
            """You can run the current file or the main file normally, or as a
script.
        When run as script, the shell is restarted to provide a clean
        environment. The shell is also initialized differently so that it
        closely resembles a normal script execution.""",
        ),
        translate(
            "wizard",
            """In interactive mode, sys.path[0] is an empty string (i.e. the
current dir),
        and sys.argv is set to [''].""",
        ),
        translate(
            "wizard",
            """In script mode, __file__ and sys.argv[0] are set to the scripts
filename,
        sys.path[0] and the working dir are set to the directory containing the
script.""",
        ),
    ]
class ToolsWizardPage1(BasePyzoWizardPage):
    _title = translate("wizard", "Tools for your convenience")
    _image_filename = "pyzo_tools1.png"
    _descriptions = [
        translate(
            "wizard",
            """Via the *Tools menu*, one can select which tools to use. The
tools can
        be positioned in any way you want, and can also be un-docked.""",
        ),
        translate(
            "wizard",
            """Note that the tools system is designed such that it's easy to
        create your own tools. Look at the online wiki for more information,
        or use one of the existing tools as an example.""",
        ),
    ]
class ToolsWizardPage2(BasePyzoWizardPage):
    _title = translate("wizard", "Recommended tools")
    _image_filename = "pyzo_tools2.png"
    _descriptions = [
        translate("wizard", """We especially recommend the following tools:"""),
        translate(
            "wizard",
            """The *Source structure tool* gives an outline of the source
code.""",
        ),
        translate(
            "wizard",
            """The *File browser tool* helps keep an overview of all files
        in a directory. To manage your projects, click the star icon.""",
        ),
    ]
class FinalPage(BasePyzoWizardPage):
    _title = translate("wizard", "Get coding!")
    _image_filename = "pyzologo128.png"
    _descriptions = [
        translate(
            "wizard",
            """This concludes the Pyzo wizard. Now, get coding and have fun!""",
        ),
    ]
# def smooth_images():
#     """ This was used to create the images from their raw versions.
#     """
#
```

```
#     import os
#     import visvis as vv
#     import scipy as sp
#     import scipy.ndimage
#     for fname in os.listdir('images'):
#         im = vv.imread(os.path.join('images', fname))
#         for i in range(im.shape[2]):
#             im[:,:,i] = sp.ndimage.gaussian_filter(im[:,:,i], 0.7)
#         #fname = fname.replace('.png', '.jpg')
#         print(fname)
#         vv.imwrite(fname, im[::2,::2,:])
if __name__ == "__main__":
    w = PyzoWizard(None)
    w.show()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2016, Almar Klein
"""
Reading and saving Zoof Object Notation files. ZON is like JSON, but a
more Pythonic format. It's just about 500 lines of code.
This format is a spin-off from the SSDF format, it is fully compatible,
except that ZON does not support numpy arrays.
"""
import re
import sys
import time
# From six.py
if sys.version_info[0] >= 3:
    string_types = (str,)
    integer_types = (int,)
else:
    string_types = (basestring,)  # noqa
    integer_types = (int, long)  # noqa
float_types = (float,)
## Dict class
try:  # pragma: no cover
    from collections import OrderedDict as _dict
except ImportError:
    _dict = dict
def isidentifier(s):
    # http://stackoverflow.com/questions/2544972/
    if not isinstance(s, str):
        return False
    return re.match(r"^\w+$", s, re.UNICODE) and re.match(r"^[0-9]", s) is None
class Dict(_dict):
    """A dict in which the items can be get/set as attributes."""
    __reserved_names__ = dir(_dict())  # Also from OrderedDict
    __pure_names__ = dir(dict())
    __slots__ = []
    def __repr__(self):
        identifier_items = []
        nonidentifier_items = []
        for key, val in self.items():
            if isidentifier(key):
                identifier_items.append("%s=%r" % (key, val))
            else:
                nonidentifier_items.append("(%r, %r)" % (key, val))
        if nonidentifier_items:
            return "Dict([%s], %s)" % (
                ", ".join(nonidentifier_items),
                ", ".join(identifier_items),
            )
        else:
            return "Dict(%s)" % (", ".join(identifier_items))
    def __getattribute__(self, key):
        try:
            return object.__getattribute__(self, key)
        except AttributeError:
            if key in self:
                return self[key]
            else:
                raise
    def __setattr__(self, key, val):
        if key in Dict.__reserved_names__:
            # Either let OrderedDict do its work, or disallow
            if key not in Dict.__pure_names__:
                return _dict.__setattr__(self, key, val)
            else:
                raise AttributeError(
                    "Reserved name, this key can only "
                    + "be set via ``d[%r] = X``" % key
                )
        else:
            # if isinstance(val, dict): val = Dict(val) -> no, makes a copy!
            self[key] = val
    def __dir__(self):
        names = [k for k in self.keys() if isidentifier(k)]
        return Dict.__reserved_names__ + names
# SSDF compatibility
Struct = Dict
Struct.__is_ssdf_struct__ = True
## Public functions
def isstruct(ob):  # SSDF compatibility
    """isstruct(ob)
    Returns whether the given object is an SSDF struct.
    """
    if hasattr(ob, "__is_ssdf_struct__"):
        return bool(ob.__is_ssdf_struct__)
    else:
        return False
def new():
    """new()
    Create a new Dict object. The same as "Dict()".
    """
    return Dict()
def clear(d):  # SSDF compatibility
    """clear(d)
    Clear all elements of the given Dict object.
    """
    d.clear()
def copy(object):
    """copy(objec)
    Return a deep copy the given object. The object and its children
    should be dict-compatible data types. Note that dicts are converted
    to Dict and tuples to lists.
    """
    if isstruct(object) or isinstance(object, dict):
        newObject = Dict()
        for key in object:
            val = object[key]
            newObject[key] = copy(val)
        return newObject
    elif isinstance(object, (tuple, list)):
        return [copy(ob) for ob in object]
    else:
        return object  # immutable
def count(object, cache=None):
    """count(object):
    Count the number of elements in the given object. An element is
    defined as one of the 6 datatypes supported by ZON (dict,
    tuple/list, string, int, float, None).
    """
    cache = cache or []
    if isstruct(object) or isinstance(object, (dict, list)):
        if id(object) in cache:
            raise RuntimeError("recursion!")
        cache.append(id(object))
    n = 1
    if isstruct(object) or isinstance(object, dict):
        for key in object:
            val = object[key]
            n += count(val, cache)
    elif isinstance(object, (tuple, list)):
        for val in object:
            n += count(val, cache)
    return n
def loads(text):
    """loads(text)
    Load a Dict from the given Unicode) string in ZON syntax.
    """
    if not isinstance(text, string_types):
        raise ValueError("zon.loads() expects a string.")
    reader = ReaderWriter()
    return reader.read(text)
def load(file):
    """load(filename)
    Load a Dict from the given file or filename.
    """
    if isinstance(file, string_types):
        file = open(file, "rb")
    text = file.read().decode("utf-8")
    return loads(text)
def saves(d):
    """saves(d)
    Serialize the given dict to a (Unicode) string.
    """
    if not (isstruct(d) or isinstance(d, dict)):
        raise ValueError("ssdf.saves() expects a dict.")
    writer = ReaderWriter()
    text = writer.save(d)
    return text
def save(file, d):
    """save(file, d)
    Serialize the given dict to the given file or filename.
    """
    text = saves(d)
    if isinstance(file, string_types):
        file = open(file, "wb")
    with file:
        file.write(text.encode("utf-8"))
## The core
class ReaderWriter(object):
    def read(self, text):
        indent = 0
        root = Dict()
        container_stack = [(0, root)]
        new_container = None
        for i, line in enumerate(text.splitlines()):
            linenr = i + 1
            # Strip line
            line2 = line.lstrip()
            # Skip comments and empty lines
            if not line2 or line2[0] == "#":
                continue
            # Find the indentation
            prev_indent = indent
            indent = len(line) - len(line2)
            if indent == prev_indent:
                pass
            elif indent < prev_indent:
                while container_stack[-1][0] > indent:
                    container_stack.pop(-1)
                if container_stack[-1][0] != indent:
                    print("ZON: Ignoring wrong dedentation at %i" % linenr)
            elif indent > prev_indent and new_container is not None:
                container_stack.append((indent, new_container))
                new_container = None
            else:
                print("ZON: Ignoring wrong indentation at %i" % linenr)
                indent = prev_indent
            # Split name and data using a regular expression
            m = re.search("^\w+? *?=", line2)
            if m:
                i = m.end(0)
                name = line2[: i - 1].strip()
                data = line2[i:].lstrip()
            else:
                name = None
                data = line2
            # Get value
            value = self.to_object(data, linenr)
            # Store the value
            _indent, current_container = container_stack[-1]
            if isinstance(current_container, dict):
                if name:
                    current_container[name] = value
                else:
                    print("ZON: unnamed item in dict on line %i" % linenr)
            elif isinstance(current_container, list):
                if name:
                    print("ZON: named item in list on line %i" % linenr)
                else:
                    current_container.append(value)
            else:
                raise RuntimeError("Invalid container %r" % current_container)
            # Prepare for next round
            if isinstance(value, (dict, list)):
                new_container = value
        return root
    def save(self, d):
        pyver = "%i.%i.%i" % sys.version_info[:3]
        ct = time.asctime()
        lines = []
        lines.append("# -*- coding: utf-8 -*-")
        lines.append("# This Zoof Object Notation (ZON) file was")
        lines.append("# created from Python %s on %s.\n" % (pyver, ct))
        lines.append("")
        lines.extend(self.from_dict(d, -2)[1:])
        return "\r\n".join(lines)
        # todo: pop toplevel dict
    def from_object(self, name, value, indent):
        # Get object's data
        if value is None:
            data = "Null"
        elif isinstance(value, integer_types):
            data = self.from_int(value)
        elif isinstance(value, float_types):
            data = self.from_float(value)
        elif isinstance(value, bool):
            data = self.from_int(int(value))
        elif isinstance(value, string_types):
            data = self.from_unicode(value)
        elif isinstance(value, dict):
            data = self.from_dict(value, indent)
        elif isinstance(value, (list, tuple)):
            data = self.from_list(value, indent)
        else:
            # We do not know
            data = "Null"
            tmp = repr(value)
            if len(tmp) > 64:
                tmp = tmp[:64] + "..."
            if name is not None:
                print("ZON: %s is unknown object: %s" % (name, tmp))
            else:
                print("ZON: unknown object: %s" % tmp)
        # Finish line (or first line)
        if isinstance(data, string_types):
            data = [data]
        if name:
            data[0] = "%s%s = %s" % (" " * indent, name, data[0])
        else:
            data[0] = "%s%s" % (" " * indent, data[0])
        return data
    def to_object(self, data, linenr):
        data = data.lstrip()
        # Determine what type of object we're dealing with by reading
        # like a human.
        if not data:
            print("ZON: no value specified at line %i." % linenr)
        elif data[0] in "-.0123456789":
            return self.to_int_or_float(data, linenr)
        elif data[0] == "'":
            return self.to_unicode(data, linenr)
        elif data.startswith("dict:"):
            return self.to_dict(data, linenr)
        elif data.startswith("list:") or data[0] == "[":
            return self.to_list(data, linenr)
        elif data.startswith("Null") or data.startswith("None"):
            return None
        else:
            print("ZON: invalid type on line %i." % linenr)
            return None
    def to_int_or_float(self, data, linenr):
        line = data.partition("#")[0]
        try:
            return int(line)
        except ValueError:
            try:
                return float(line)
            except ValueError:
                print("ZON: could not parse number on line %i." % linenr)
                return None
    def from_int(self, value):
        return repr(int(value)).rstrip("L")
    def from_float(self, value):
        # Use general specifier with a very high precision.
        # Any spurious zeros are automatically removed. The precision
        # should be sufficient such that any numbers saved and loaded
        # back will have the exact same value again.
        # see e.g. http://bugs.python.org/issue1580
        return repr(float(value))  # '%0.17g' % value
    def from_unicode(self, value):
        value = value.replace("\\", "\\\\")
        value = value.replace("\n", "\\n")
        value = value.replace("\r", "\\r")
        value = value.replace("\x0b", "\\x0b").replace("\x0c", "\\x0c")
        value = value.replace("'", "\\'")
        return "'" + value + "'"
    def to_unicode(self, data, linenr):
        # Encode double slashes
        line = data.replace("\\\\", "0x07")  # temp
        # Find string using a regular expression
        m = re.search("'.*?[^\\\\]'|''", line)
        if not m:
            print("ZON: string not ended correctly on line %i." % linenr)
            return None  # return not-a-string
        else:
            line = m.group(0)[1:-1]
        # Decode stuff
        line = line.replace("\\n", "\n")
        line = line.replace("\\r", "\r")
        line = line.replace("\\x0b", "\x0b").replace("\\x0c", "\x0c")
        line = line.replace("\\'", "'")
        line = line.replace("0x07", "\\")
        return line
    def from_dict(self, value, indent):
        lines = ["dict:"]
        # Process children
        for key, val in value.items():
            # Skip all the builtin stuff
            if key.startswith("__"):
                continue
            # Skip methods, or anything else we can call
            if hasattr(val, "__call__"):
                continue  # Note: py3.x does not have function callable
            # Add!
            lines.extend(self.from_object(key, val, indent + 2))
        return lines
    def to_dict(self, data, linenr):
        return Dict()
    def from_list(self, value, indent):
        # Collect subdata and check whether this is a "small list"
        isSmallList = True
        allowedTypes = integer_types + float_types + string_types
        subItems = []
        for element in value:
            if not isinstance(element, allowedTypes):
                isSmallList = False
            subdata = self.from_object(None, element, 0)  # No indent
            subItems.extend(subdata)
        isSmallList = isSmallList and len(subItems) < 256
        # Return data
        if isSmallList:
            return "[%s]" % (", ".join(subItems))
        else:
            data = ["list:"]
            ind = " " * (indent + 2)
            for item in subItems:
                data.append(ind + item)
            return data
    def to_list(self, data, linenr):
        value = []
        if data[0] == "l":  # list:
            return list()
        else:
            i0 = 1
            pieces = []
            inString = False
            escapeThis = False
            line = data
            for i in range(1, len(line)):
                if inString:
                    # Detect how to get out
                    if escapeThis:
                        escapeThis = False
                        continue
                    elif line[i] == "\\":
                        escapeThis = True
                    elif line[i] == "'":
                        inString = False
                else:
```

```
                # Detect going in a string, break, or end
                if line[i] == "'":
                    inString = True
                elif line[i] == ",":
                    pieces.append(line[i0:i])
                    i0 = i + 1
                elif line[i] == "]":
                    piece = line[i0:i]
                    if piece.strip():  # Do not add if empty
                        pieces.append(piece)
                    break
        else:
            print("ZON: short list not closed right on line %i." % linenr)
        # Cut in pieces and process each piece
        value = []
        for piece in pieces:
            v = self.to_object(piece, linenr)
            value.append(v)
        return value
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Pyzo is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" pyzo.util._locale
Module for locale stuff like language and translations.
"""
import os, sys, time
import pyzo
from pyzo.qt import QtCore, QtWidgets
QLocale = QtCore.QLocale
# Define supported languages. The key defines the name as shown to the
# user. The value is passed to create a Locale object. From the local
# object we obtain the name for the .tr file.
# Chinese:
LANGUAGES = {
    "English (US)": QLocale.C,
    # == (QLocale.English, QLocale.UnitedStates),
    #'English (UK)': (QLocale.English, QLocale.UnitedKingdom),
    "Dutch": QLocale.Dutch,
    "Spanish": QLocale.Spanish,
    "Catalan": QLocale.Catalan,
    "French": QLocale.French,
    "German": QLocale.German,
    "Russian": QLocale.Russian,  # not updated for 3.4
    "Polish": QLocale.Polish,
    "Portuguese": QLocale.Portuguese,
    "Portuguese (BR)": (QLocale.Portuguese, QLocale.Brazil),
    "Simplified Chinese": QLocale.Chinese,
    "Traditional Chinese": (
        QLocale.Chinese,
        QLocale.Taiwan,
    ),  # https://bugreports.qt.io/browse/QTBUG-1573
    # Languages for which the is a .tr file, but no translations available yet:
    # 'Slovak': QLocale.Slovak,
}
LANGUAGE_SYNONYMS = {
    None: "English (US)",
    "": "English (US)",
    "English": "English (US)",
    "ca_ES": "Catalan",
    "de_DE": "German",
    "es_ES": "Spanish",
    "fr_FR": "French",
    "nl_NL": "Dutch",
    "pl_PL": "Polish",
    "pt_BR": "Portuguese (BR)",
    "pt_PT": "Portuguese",
    "ru_RU": "Russian",
    "zh_CN": "Simplified Chinese",
    "zh_TW": "Traditional Chinese",
}
def getLocale(languageName):
    """getLocale(languageName)
    Get the QLocale object for the given language (as a string).
    """
    # Try System  Language if nothing defined
    if languageName == "":
        languageName = QLocale.system().name()
    # Apply synonyms
    languageName = LANGUAGE_SYNONYMS.get(languageName, languageName)
    # if no language applicable, get back to default
    if LANGUAGES.get(languageName, None) is None:
        languageName = LANGUAGE_SYNONYMS.get("", "")
    # Select language in qt terms
    qtLanguage = LANGUAGES.get(languageName, None)
    if qtLanguage is None:
        raise ValueError("Unknown language")
    # Return locale
    if isinstance(qtLanguage, tuple):
        return QLocale(*qtLanguage)
    else:
        return QLocale(qtLanguage)
def setLanguage(languageName):
    """setLanguage(languageName)
    Set the language for the app. Loads qt and pyzo translations.
    Returns the QLocale instance to pass to the main widget.
    """
    # Get locale
    locale = getLocale(languageName)
    # Get paths were language files are
    qtTransPath = str(
        QtCore.QLibraryInfo.location(QtCore.QLibraryInfo.TranslationsPath)
    )
    pyzoTransPath = os.path.join(pyzo.pyzoDir, "resources", "translations")
    # Get possible names for language files
    # (because Qt's .tr files may not have the language component.)
    localeName1 = locale.name()
    localeName2 = localeName1.split("_")[0]
    # Uninstall translators
    if not hasattr(QtCore, "_translators"):
        QtCore._translators = []
    for trans in QtCore._translators:
        QtWidgets.QApplication.removeTranslator(trans)
    # The default language
    if localeName1 == "C":
        return locale
    # Set Qt translations
    # Note that the translator instances must be stored
    # Note that the load() method is very forgiving with the file name
    for what, where in [("qt", qtTransPath), ("pyzo", pyzoTransPath)]:
        trans = QtCore.QTranslator()
        # Try loading both names
        for localeName in [localeName1, localeName2]:
            success = trans.load(what + "_" + localeName + ".tr", where)
            if success:
                QtWidgets.QApplication.installTranslator(trans)
                QtCore._translators.append(trans)
                print("loading %s %s: ok" % (what, languageName))
                break
        else:
            print("loading %s %s: failed" % (what, languageName))
    # Done
    return locale
class Translation(str):
    """Derives from str class. The translate function returns an instance
    of this class and assigns extra atrributes:
      * original: the original text passed to the translation
      * tt: the tooltip text
      * key: the original text without tooltip (used by menus as a key)
    We adopt a simple system to include tooltip text in the same
    translation as the label text. By including ":::" in the text,
    the text after that identifier is considered the tooltip.
    The text returned by the translate function is always the
    string without tooltip, but the text object has an attribute
    "tt" that stores the tooltip text. In this way, if you do not
    use this feature or do not know about this feature, everything
    keeps working as expected.
    """
    pass
def _splitMainAndTt(s):
    if ":::" in s:
        parts = s.split(":::", 1)
        return parts[0].rstrip(), parts[1].lstrip()
    else:
        return s, ""
def translate(context, text, disambiguation=None):
    """translate(context, text, disambiguation=None)
    The translate function used throughout pyzo.
    """
    # Get translation and split tooltip
    newtext = QtCore.QCoreApplication.translate(context, text, disambiguation)
    s, tt = _splitMainAndTt(newtext)
    # Create translation object (string with extra attributes)
    translation = Translation(s)
    translation.original = text
    translation.tt = tt
    translation.key = _splitMainAndTt(text)[0].strip()
    return translation
## Development tools
import subprocess
LHELP = """
Language help - info for translaters
For translating, you will need a set of working Qt language tools:
pyside-lupdate, linguist, lrelease. On Windows, these should come
with your PySide installation. On (Ubuntu) Linux, you can install
these with 'sudo apt-get install pyside-tools qt4-dev-tools'.
You also need to run pyzo from source as checked out from the repo
(e.g. by running pyzolauncher.py).
To create a new language:
  * the file 'pyzo/util/locale.py' should be edited to add the language
    to the LANGUAGES dict
  * run 'linguist(your_lang)', this will raise an erro, but it will show
    the name of the .tr file
  * the file 'pyzo/pyzo.pro' should be edited to include the new .tr file
  * run 'lupdate()' to create the .tr file
  * run 'linguist(your_lang)' again to initialize the .tr file.
To update a language:
  * run 'lupdate()'
  * run 'linguist(your_lang)'
  * make all the translations and save
  * run lrelease() and restart pyzo to see translations
  * repeat if necessary
"""
def lhelp():
    """lhelp()
    Print help text on using the language tools.
    """
    print(LHELP)
def linguist(languageName):
    """linguist(languageName)
    Open linguist with the language file as specified by lang. The
    languageName can be one of the fields as visible in the language
    list in the menu. This function is intended for translators.
    """
    # Get locale
    locale = getLocale(languageName)
    # Get file to open
    fname = "pyzo_{}.tr".format(locale.name())
    filename = os.path.join(pyzo.pyzoDir, "resources", "translations", fname)
    if not os.path.isfile(filename):
        raise ValueError("Could not find {}".format(filename))
    # PyQt5 does not come with linguist anymore? Install PySide2 and check the
    # pyside2 package directory for the linguist exe ...
    # Get Command for linguist
    qtcore_mod_name = pyzo.QtCore.QObject.__module__
    qtcore_mod_path = sys.modules[qtcore_mod_name].__file__
    pysideDir = os.path.abspath(os.path.dirname(qtcore_mod_path))
    print(pysideDir)
    ISWIN = sys.platform.startswith("win")
    exe_ = "linguist" + ".exe" * ISWIN
    exe = os.path.join(pysideDir, exe_)
    if not os.path.isfile(exe):
        exe = exe_
    # Spawn process
    return subprocess.Popen([exe, filename])
def lupdate():
    """For developers. From pyzo.pro create the .tr files"""
    # Get file to open
    fname = "pyzo.pro"
    filename = os.path.realpath(os.path.join(pyzo.pyzoDir, "..", fname))
    if not os.path.isfile(filename):
        raise ValueError(
            "Could not find {}. This function must run from the source
repo.".format(
                fname
            )
        )
    # Get Command for python lupdate
    pysideDir = os.path.abspath(os.path.dirname(pyzo.QtCore.__file__))
    ISWIN = sys.platform.startswith("win")
    exe_ = "pylupdate" + pyzo.QtCore.__version__[0] + ".exe" * ISWIN
    exe = os.path.join(pysideDir, exe_)
    if not os.path.isfile(exe):
        exe = exe_
    # Spawn process
    cmd = [exe, "-noobsolete", "-verbose", filename]
    p = subprocess.Popen(
        cmd, shell=False, stdout=subprocess.PIPE, stderr=subprocess.STDOUT
    )
    while p.poll() is None:
        time.sleep(0.1)
    output = p.stdout.read().decode("utf-8")
    if p.returncode:
        raise RuntimeError("lupdate failed (%i): %s" % (p.returncode, output))
    else:
        print(output)
def lrelease():
    """For developers. From pyzo.pro and the .tr files, create the .qm files."""
    # Get file to open
    fname = "pyzo.pro"
    filename = os.path.realpath(os.path.join(pyzo.pyzoDir, "..", fname))
    if not os.path.isfile(filename):
        raise ValueError(
            "Could not find {}. This function must run from the source
repo.".format(
                fname
            )
        )
    # Get Command for lrelease
    pysideDir = os.path.abspath(os.path.dirname(pyzo.QtCore.__file__))
    ISWIN = sys.platform.startswith("win")
    exe_ = "lrelease" + ".exe" * ISWIN
    exe = os.path.join(pysideDir, exe_)
    if not os.path.isfile(exe):
        exe = exe_
    # Spawn process
    cmd = [exe, filename]
    p = subprocess.Popen(
        cmd, shell=False, stdout=subprocess.PIPE, stderr=subprocess.STDOUT
    )
    while p.poll() is None:
        time.sleep(0.1)
    output = p.stdout.read().decode("utf-8")
    if p.returncode:
        raise RuntimeError("lrelease failed (%i): %s" % (p.returncode, output))
    else:
        print(output)
if __name__ == "__main__":
    # Print names of translator files
    print("Language data files:")
    for key in LANGUAGES:
        s = "{}: {}".format(key, getLocale(key).name() + ".tr")
        print(s)
```

```
# -*- coding: utf-8 -*-
# Copyright (c) 2016, Almar Klein
"""
This module implements functionality to obtain registered
Python interpreters and to register a Python interpreter in the Windows
registry.
"""
import sys
import os
try:
    import winreg
except ImportError:
    winreg = None
PYTHON_KEY = "SOFTWARE\\Python\\PythonCore"
PYTHON_KEY_WOW64 = "SOFTWARE\\Wow6432Node\\Python\\PythonCore"
INSTALL_KEY = "InstallPath"
PATH_KEY = "PythonPath"
class PythonInReg:
    """Class to represent a specific version of the Python interpreter
    registered (or being registered in the registry).
    This is a helper class for the functions defined in this module; it
    should not be instantiated directly.
    """
    USER_ONE = 1
    USER_ALL = 2
    def __init__(self, user, version, wow64=False):
        self._user = user
        self._key = (wow64 and PYTHON_KEY_WOW64 or PYTHON_KEY) + "\\" + version
    def __repr__(self):
        userstr = [None, "USER_ONE", "USER_ALL"][self._user]
        installPath = self.installPath()
        reg = self._reg()
        if not reg:
            return "<PythonInReg %s at %s (unregistered)>" % (self.version(),
userstr)
        elif installPath:
            return '<PythonInReg %s at %s in "%s">' % (
                self.version(),
                userstr,
                installPath,
            )
        else:
            return "<PythonInReg %s at %s>" % (self.version(), userstr)
    def _root(self):
        if self._user == PythonInReg.USER_ONE:
            return winreg.HKEY_CURRENT_USER
        else:
            return winreg.HKEY_LOCAL_MACHINE
    def _reg(self):
        # Get key for this version
        try:
            return winreg.OpenKey(self._root(), self._key, 0, winreg.KEY_READ)
        except Exception:
            return None
    def create(self):
        """Create key. If already exists, does nothing."""
        # Get key for this version
        reg = self._reg()
        if reg:
            winreg.CloseKey(reg)
            # print('Unable to create Python version %s: already exists.' %
self.version())
        else:
            # Try to create
            try:
                reg = winreg.CreateKey(self._root(), self._key)
                winreg.CloseKey(reg)
            except Exception:
                raise RuntimeError(
                    "Unable to create python version %s." % self.version()
                )
            print("Created %s." % str(self))
    def delete(self):
        # Get key for this version
        reg = self._reg()
        if not reg:
            print("Unable to delete Python version %s: does not exist.")
        # Delete attributes
        try:
            winreg.DeleteKey(reg, INSTALL_KEY)
        except Exception:
            pass
        try:
            winreg.DeleteKey(reg, PATH_KEY)
        except Exception:
            pass
        # Delete main key for this version, or show warning
        try:
            winreg.DeleteKey(self._root(), self._key)
        except Exception:
            print("Could not delete %s." % str(self))
            return
        print("Deleted %s." % str(self))
    def setInstallPath(self, installPath):
        # Get key for this version
        reg = self._reg()
        if not reg:
            raise RuntimeError(
                "Could not set installPath for version %s: version does not
exist."
                % self.version()
            )
        # Set value or raise error
        try:
            winreg.SetValue(reg, INSTALL_KEY, winreg.REG_SZ, installPath)
            winreg.CloseKey(reg)
        except Exception:
            winreg.CloseKey(reg)
            raise RuntimeError("Could not set installPath for %s." % str(self))
    def installPath(self):
        # Get key for this version
        reg = self._reg()
        if not reg:
            return None
        # Get value or return None
        try:
            installPath = winreg.QueryValue(reg, INSTALL_KEY)
            winreg.CloseKey(reg)
            return installPath
        except Exception:
            winreg.CloseKey(reg)
            return None
    def setPythonPath(self, pythonPath):
        # Get key for this version
        reg = self._reg()
        if not reg:
            raise RuntimeError(
                "Could not set pythonPath for version %s: version does not
exist."
```

```
                % self.version()
            )
        # Set value or raise error
        try:
            winreg.SetValue(reg, PATH_KEY, winreg.REG_SZ, pythonPath)
            winreg.CloseKey(reg)
        except Exception:
            winreg.CloseKey(reg)
            raise RuntimeError("Could not set pythonPath for %s." % str(self))
    def pythonPath(self):
        # Get key for this version
        reg = self._reg()
        if not reg:
            return None
        # Get value or return None
        try:
            pythonPath = winreg.QueryValue(reg, PATH_KEY)
            winreg.CloseKey(reg)
            return pythonPath
        except Exception:
            winreg.CloseKey(reg)
            return None
    def version(self):
        """Get the Python version."""
        return self._key[-3:]
def get_interpreters_in_reg():
    """get_interpreters_in_reg()
    Get a list of PythonInReg instances: one for each interpreter
    in the registry. This function checks both LOCAL_MACHINE and CURRENT_USER.
    """
    versions = []
    for user in [1, 2]:
        for wow64 in [False, True]:
            versions.extend(_get_interpreter_in_reg(user, wow64))
    return versions
def _get_interpreter_in_reg(user, wow64=False):
    # Get base key
    if user == PythonInReg.USER_ONE:
        HKEY = winreg.HKEY_CURRENT_USER
    else:
        HKEY = winreg.HKEY_LOCAL_MACHINE
    # Get Python key
    if wow64:
        PYKEY = PYTHON_KEY_WOW64
    else:
        PYKEY = PYTHON_KEY
    # Try to open Python key
    try:
        reg = winreg.OpenKey(HKEY, PYKEY, 0, winreg.KEY_READ)
    except Exception:
        return []
    # Get info about subkeys
    nsub, nval, modified = winreg.QueryInfoKey(reg)
    # Query all
    versions = []
    for i in range(nsub):
        # Get name and subkey
        version = winreg.EnumKey(reg, i)
        versions.append(PythonInReg(user, version, wow64))
    # Done
    winreg.CloseKey(reg)
    return versions
def register_interpreter(version=None, installPath=None, user=None,
wow64=False):
    """register_interpreter(version=None, installPath=None, user=None,
wow64=False)
    Register a certain Python version. If version and installPath
    are not given, the current Python process is registered.
    if user is not given, tries LOCAL_MACHINE first but uses CURRENT_USER
    if that fails.
    """
    if version is None:
        version = sys.version[:3]
    if installPath is None:
        installPath = sys.prefix
    # Get existing versions
    existingVersions = get_interpreters_in_reg()
    # Determine what users to try
    if user is None:
        users = [2, 1]
    else:
        users = [user]
    success = False
    for user in users:
        # Create new PythonInReg instance
        v = PythonInReg(user, version, wow64)
        # Check if already exists
        ok = True
        for ev in existingVersions:
            if ev._key != v._key or ev._user != v._user:
                continue  # Different key; no problem
            if (not ev.installPath()) or (not os.path.isdir(ev.installPath())):
                continue  # Key the same, but existing entry is invalid
            if ev.installPath() == installPath:
                # Exactly the same, no action required, return now!
                return ev
            # Ok, there's a problem
            ok = False
            print(
                'Warning: version %s is already installed in "%s".'
                % (version, ev.installPath())
            )
        if not ok:
            continue
        # Try to create the key
        try:
            v.create()
            v.setInstallPath(installPath)
            success = True
            break
        except RuntimeError:
            continue
    if success:
        return v
    else:
        raise RuntimeError(
            "Could not register Python version %s at %s." % (version,
installPath)
        )
if __name__ == "__main__":
    for v in get_interpreters_in_reg():
        print(v)
```

```
import os
import sys
import subprocess
from .inwinreg import register_interpreter
EXE_DIR = os.path.abspath(os.path.dirname(sys.executable))
if EXE_DIR.endswith(".app/Contents/MacOS"):
    EXE_DIR = os.path.dirname(EXE_DIR.rsplit(".app")[0])
def make_abs(path):
    if path.startswith("."):
        return os.path.abspath(os.path.join(EXE_DIR, path))
    return path
class PythonInterpreter:
    """Class to represent a Python interpreter. It has properties
    to get the path and version. Upon creation the version number is
    acquired by calling the interpreter in a subprocess. If this fails,
    the version becomes ''.
    """
    def __init__(self, path):
        if not isinstance(path, str):
            raise ValueError("Path for PythonInterpreter is not a string: %r" %
path)
        if not os.path.isfile(make_abs(path)):
            raise ValueError("Path for PythonInterpreter is invalid: %r" % path)
        self._path = (
            path if path.startswith(".") else
os.path.normpath(os.path.abspath(path))
        )
        self._normpath = os.path.normcase(self._path)
        self._problem = ""
        self._version = None
        # Set prefix
        self._prefix = os.path.dirname(self.path)
        if os.path.basename(self._prefix) == "bin":
            self._prefix = os.path.dirname(self._prefix)
    def __repr__(self):
        cls_name = self.__class__.__name__
        return "<%s version %s at %s>" % (cls_name, self.version, self.path)
    def __hash__(self):
        return hash(self._normpath)
    def __eq__(self, other):
        return self._normpath == other._normpath
    @property
    def path(self):
        """The path to the executable of the Python interpreter.
        If relative (starting with a dot), it is relative to the current
        sys.executable.
        """
        return self._path
    @property
    def prefix(self):
        """The prefix of this executable."""
        return self._prefix
    @property
    def is_conda(self):
        """Whether this interpreter is part of a conda environment (either
        a root or an env).
        """
        return os.path.isdir(os.path.join(make_abs(self._prefix), "conda-meta"))
    @property
    def version(self):
        """The version number as a string, usually 3 numbers."""
        if self._version is None:
            self._version = self._getversion()
        return self._version
    @property
    def version_info(self):
        """The version number as a tuple of integers. For comparing."""
        return versionStringToTuple(self.version)
    def register(self):
        """Register this Python intepreter. On Windows this modifies
        the CURRENT_USER. On All other OS's this is a no-op.
        """
        if sys.platform.startswith("win"):
            path = os.path.split(make_abs(self.path))[0]  # Remove "python.exe"
            register_interpreter(self.version[:3], path)
    def _getversion(self):
        path = make_abs(self._path)
        # Check if path is even a file
        if not os.path.isfile(path):
            self._problem = "%s is not a valid file."
            return ""
        # Poll Python executable (--version does not work on 2.4)
        # shell=True prevents loads of command windows popping up on Windows,
        # but if used on Linux it would enter interpreter mode
        cmd = [path, "-V"]
        try:
            v = subprocess.check_output(
                cmd, stderr=subprocess.STDOUT,
shell=sys.platform.startswith("win")
            )
        except Exception as e:  # Don't risk not catching an unforeseen
exception ...
            self._problem = str(e)
            return ""
        # Extract the version, apply some defensive programming
        v = v.decode("ascii", "ignore").strip().lower()
        if v.startswith("python"):
            v = v.split(" ")[1]
        v = v.split(" ")[0]
        # Try turning it into version_info
        try:
            versionStringToTuple(v)
        except ValueError:
            return ""
        # Done
        return v
def versionStringToTuple(version):
    # Truncate version number to first occurance of non-numeric character
    tversion = ""
    for c in version:
        if c in "0123456789.":
            tversion += c
    # Split by dots, make each number an integer
    tversion = tversion.strip(".")
    return tuple([int(a) for a in tversion.split(".") if a])
```

```
# -*- coding: utf-8 -*-
# Copyright (c) 2016, Almar Klein
"""
This module implements functionality to detect available Python
interpreters. This is done by looking at common locations, the Windows
registry, and conda's environment list.
"""
import sys
import os
from .pythoninterpreter import EXE_DIR, PythonInterpreter, versionStringToTuple
from .inwinreg import get_interpreters_in_reg
from .. import paths
def get_interpreters(minimumVersion=None):
    """get_interpreters(minimumVersion=None)
    Returns a list of PythonInterpreter instances.
    If minimumVersion is given, return only the interprers with at least that
    version, and also sort the result by version number.
    """
    # Get Python interpreters
    if sys.platform.startswith("win"):
        pythons = _get_interpreters_win()
    else:
        pythons = _get_interpreters_posix()
    pythons = set([PythonInterpreter(p) for p in pythons])
    # Get conda paths
    condas = set([PythonInterpreter(p) for p in _get_interpreters_conda()])
    # Get relative interpreters
    relative = set([PythonInterpreter(p) for p in _get_interpreters_relative()])
    # Get Pyzo paths
    pyzos = set([PythonInterpreter(p) for p in _get_interpreters_pyzo()])
    # Get pipenv paths
    pipenvs = set([PythonInterpreter(p) for p in _get_interpreters_pipenv()])
    # Almost done
    interpreters = set.union(pythons, condas, relative, pyzos, pipenvs)
    minimumVersion = minimumVersion or "0"
    return _select_interpreters(interpreters, minimumVersion)
def _select_interpreters(interpreters, minimumVersion):
    """Given a list of PythonInterpreter instances, return a list with
    the interpreters selected that are valid and have their version equal
    or larger than the given minimimVersion. The returned list is sorted
    by version number.
    """
    if not isinstance(minimumVersion, str):
        raise ValueError("minimumVersion in get_interpreters must be a string.")
    # Remove invalid interpreters
    interpreters = [i for i in interpreters if i.version]
    # Remove the ones below the reference version
    if minimumVersion is not None:
        refTuple = versionStringToTuple(minimumVersion)
        interpreters = [i for i in interpreters if (i.version_info >= refTuple)]
    # Return, sorted by version
    return sorted(interpreters, key=lambda x: x.version_info)
def _get_interpreters_win():
    found = []
    # Query from registry
    for v in get_interpreters_in_reg():
        found.append(v.installPath())
    # Check common locations
    for rootname in [
        "C:/",
        "~/",
        "C:/program files/",
        "C:/program files (x86)/",
        "C:/ProgramData/",
        "~/appdata/local/programs/python/",
        "~/appdata/local/continuum/",
        "~/appdata/local/anaconda/",
    ]:
        rootname = os.path.expanduser(rootname)
        if not os.path.isdir(rootname):
            continue
        for dname in os.listdir(rootname):
            if dname.lower().startswith(("python", "pypy", "miniconda",
"anaconda")):
                found.append(os.path.join(rootname, dname))
    # remove 'None' path added in the found list (when badly registered python
in the user system)
    found = [v for v in found if v]
    # Normalize all paths, and remove trailing backslashes
    found = [os.path.normcase(os.path.abspath(v)).strip("\\") for v in found]
    # Append "python.exe" and check if that file exists
    found2 = []
    for dname in found:
        for fname in ("python.exe", "pypy.exe"):
            exename = os.path.join(dname, fname)
            if os.path.isfile(exename):
                found2.append(exename)
                break
    # Return as set (remove duplicates)
    return set(found2)
def _get_interpreters_posix():
    found = []
    def isPathValidPythonExe(filename):
        fname = os.path.split(filename)[1]
        return (
            fname.startswith(("python", "pypy"))
            and not fname.count("config")
            and len(fname) < 16
            and os.path.isfile(filename)
        )
    # Look for system Python interpreters
    for searchpath in ["/usr/bin", "/usr/local/bin", "/opt/local/bin"]:
        searchpath = os.path.expanduser(searchpath)
        # Get files
        try:
            files = os.listdir(searchpath)
        except Exception:
            continue
        # Search for python executables
        for fname in files:
            # Get filename and resolve symlink
            # Seen on OS X that was not a valid file
            filename = os.path.join(searchpath, fname)
            filename = os.path.realpath(filename)
            if isPathValidPythonExe(filename):
                found.append(filename)
    # Look for user-installed Python interpreters such as pypy and anaconda
    for rootname in ["~", "/usr/local"]:
        rootname = os.path.expanduser(rootname)
        if not os.path.isdir(rootname):
            continue
        for dname in os.listdir(rootname):
            if dname.lower().startswith(("python", "pypy", "miniconda",
"anaconda")):
                for fname in ("bin/python", "bin/pypy"):
                    exename = os.path.join(rootname, dname, fname)
                    if os.path.isfile(exename):
                        found.append(exename)
    # Look for Python interpreter provided by PYZO_DEFAULT_SHELL_PYTHON_EXE env-
```

```
var
    if "PYZO_DEFAULT_SHELL_PYTHON_EXE" in os.environ:
        filename = os.environ["PYZO_DEFAULT_SHELL_PYTHON_EXE"]
        filename = os.path.realpath(filename)
        if isPathValidPythonExe(filename):
            found.append(filename)
    # Remove pythonw, pythonm and the like
    found = set(found)
    for path in list(found):
        if path.endswith(("m", "w")) and path[:-1] in found:
            found.discard(path)
    # Return as set (remove duplicates)
    return set(found)
def _get_interpreters_pyzo():
    """Get a list of known Pyzo interpreters."""
    pythonname = "python" + ".exe" * sys.platform.startswith("win")
    exes = []
    for d in paths.pyzo_dirs():
        for fname in [
            os.path.join(d, "bin", pythonname + "3"),
            os.path.join(d, pythonname),
        ]:
            if os.path.isfile(fname):
                exes.append(fname)
                break
    return exes
def _get_interpreters_conda():
    """Get known conda environments"""
    if sys.platform.startswith("win"):
        pythonname = "python" + ".exe"
    else:
        pythonname = "bin/python"
    exes = []
    filename = os.path.expanduser("~/.conda/environments.txt")
    if os.path.isfile(filename):
        for line in open(filename, "rt").readlines():
            line = line.strip()
            exe_filename = os.path.join(line, pythonname)
            if line and os.path.isfile(exe_filename):
                exes.append(exe_filename)
    return exes
def _get_interpreters_pipenv():
    """Get known pipenv environments"""
    if sys.platform.startswith("win"):
        pythonname = "Scripts\\python" + ".exe"
    else:
        pythonname = "bin/python"
    exes = []
    for basedir in [os.path.join(os.path.expanduser("~"), ".virtualenvs")]:
        if os.path.isdir(basedir):
            for d in os.listdir(basedir):
                exe_filename = os.path.join(basedir, d, pythonname)
                if os.path.isfile(exe_filename):
                    exes.append(exe_filename)
    return exes
def _get_interpreters_relative():
    """Get interpreters relative to our prefix and the parent directory.
    This allows Pyzo to be shipped inside a pre-installed conda env.
    """
    pythonname = "python" + ".exe" * sys.platform.startswith("win")
    exes = []
    for d in [".", ".."]:
        for fname in [
            os.path.join(d, "bin", pythonname),
            os.path.join(d, pythonname),
        ]:
            filename = os.path.abspath(os.path.join(EXE_DIR, fname))
            if os.path.isfile(filename):
                exes.append(fname)
                break
    return exes
if __name__ == "__main__":
    for pi in get_interpreters():
        print(pi)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" yoton.clientserver.py
Yoton comes with a small framework to setup a request-reply pattern
using a client-server model (over a non-persistent connection),
similar to telnet. This allows one process to easily ask small pieces
of information from another process.
To create a server, create a class that inherits from
yoton.RequestServer and implement its handle_request() method.
A client process can simply use the yoton.do_request function.
Example: ``yoton.do_request('www.google.com:80', 'GET http/1.1\\r\\n')``
The client server model is implemented using one function and one class:
yoton.do_request and yoton.RequestServer.
Details
-------
The server implements a request/reply pattern by listening at a socket.
Similar to telnet, each request is handled using a connection
and the socket is closed after the response is send.
The request server can setup to run in the main thread, or can be started
using its own thread. In the latter case, one can easily create multiple
servers in a single process, that listen on different ports.
"""
import time
import socket
import threading
from yoton.misc import basestring, str
from yoton.misc import split_address, getErrorMsg
from yoton.core import send_all, recv_all
class RequestServer(threading.Thread):
    """RequestServer(address, async_val=False, verbose=0)
    Setup a simple server that handles requests similar to a telnet server,
    or asyncore. Starting the server using run() will run the server in
    the calling thread. Starting the server using start() will run the
    server in a separate thread.
    To create a server, subclass this class and re-implement the
    handle_request method. It accepts a request and should return a
    reply. This server assumes utf-8 encoded messages.
    Parameters
    ----------
    address : str
```

```
        Should be of the shape hostname:port.
    async_val : bool
        If True, handles each incoming connection in a separate thread.
        This might be advantageous if a the handle_request() method
        takes a long time to execute.
    verbose : bool
        If True, print a message each time a connection is accepted.
    Notes on hostname
    -----------------
    The hostname can be:
      * The IP address, or the string hostname of this computer.
      * 'localhost': the connections is only visible from this computer.
        Also some low level networking layers are bypassed, which results
        in a faster connection. The other context should also connect to
        'localhost'.
      * 'publichost': the connection is visible by other computers on the
        same network.
    """
    def __init__(self, address, async_val=False, verbose=0):
        threading.Thread.__init__(self)
        # Store whether to handle requests asynchronously
        self._async = async_val
        # Verbosity
        self._verbose = verbose
        # Determine host and port (assume tcp)
        protocol, host, port = split_address(address)
        # Create socket. Apply SO_REUSEADDR when binding, so that a
        # improperly closed socket on the same port will not prevent
        # us connecting.
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        # Bind (can raise error is port is not available)
        s.bind((host, port))
        # Store socket instance
        self._bsd_socket = s
        # To stop serving
        self._stop_me = False
        # Make deamon
        self.setDaemon(True)
    def start(self):
        """start()
        Start the server in a separate thread.
        """
        self._stop_me = False
        threading.Thread.start(self)
    def stop(self):
        """stop()
        Stop the server.
        """
        self._stop_me = True
    def run(self):
        """run()
        The server's main loop.
        """
        # Get socket instance
        s = self._bsd_socket
        # Notify
        hostname, port = s.getsockname()
        if self._verbose:
            print("Yoton: hosting at %s, port %i" % (hostname, port))
        # Tell the socket it is a host, accept multiple
        s.listen(1)
        # Set timeout so that we can check _stop_me from time to time
        self._bsd_socket.settimeout(0.25)
        # Enter main loop
        while not self._stop_me:
            try:
                s, addr = self._bsd_socket.accept()
            except socket.timeout:
                pass
            except InterruptedError:
                pass
            else:
                # Show handling?
                if self._verbose:
                    print("handling request from: " + str(addr))
                # Handle request
                if self._async:
                    rh = SocketHandler(self, s)
                    rh.start()
                else:
                    self._handle_connection(s)
        # Close down
        try:
            self._bsd_socket.close()
        except socket.error:
            pass
    def _handle_connection(self, s):
        """_handle_connection(s)
        Handle an incoming connection.
        """
        try:
            self._really_handle_connection(s)
        except Exception:
            print("Error handling request:")
            print(getErrorMsg())
    def _really_handle_connection(self, s):
        """_really_handle_connection(s)
        Really handle an incoming connection.
        """
        # Get request
        request = recv_all(s, True)
        if request is None:
            return
        # Get reply
        reply = self.handle_request(request)
        # Test
        if not isinstance(reply, basestring):
            raise ValueError("handle_request() should return a string.")
        # Send reply
        send_all(s, reply, True)
        # Close the socket
        try:
            s.close()
        except socket.error:
            pass
    def handle_request(self, request):
        """handle_request(request)
        Return a reply, given the request. Overload this method to create
        a server.
        De standard implementation echos the request, waits one second
        when receiving 'wait' and stop the server when receiving 'stop'.
        """
        # Special cases
        if request == "wait":
            time.sleep(1.0)
        elif request == "stop":
            self._stop_me = True
        # Echo
        return "Requested: " + request
class SocketHandler(threading.Thread):
    """SocketHandler(server, s)
    Simple thread that handles a connection.
    """
    def __init__(self, server, s):
        threading.Thread.__init__(self)
        self._server = server
        self._bsd_socket = s
    def run(self):
        self._server._handle_connection(self._bsd_socket)
def do_request(address, request, timeout=-1):
    """do_request(address, request, timeout=-1)
    Do a request at the server at the specified address. The server can
    be a yoton.RequestServer, or any other server listening on a socket
    and following a REQ/REP pattern, such as html or telnet. For example:
    ``html = do_request('www.google.com:80', 'GET http/1.1\\r\\n')``
    Parameters
    ----------
    address : str
        Should be of the shape hostname:port.
    request : string
        The request to make.
    timeout : float
        If larger than 0, will wait that many seconds for the respons, and
        return None if timed out.
    Notes on hostname
    -----------------
    The hostname can be:
      * The IP address, or the string hostname of this computer.
      * 'localhost': the connections is only visible from this computer.
        Also some low level networking layers are bypassed, which results
        in a faster connection. The other context should also connect to
        'localhost'.
      * 'publichost': the connection is visible by other computers on the
        same network.
    """
    # Determine host (assume tcp)
    protocol, host, port = split_address(address)
    # Check request
    if not isinstance(request, basestring):
        raise ValueError("request should be a string.")
    # Check timeout
    if timeout is None:
        timeout = -1
    # Create socket and connect
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        s.connect((host, port))
    except socket.error:
        raise RuntimeError("No server is listening at the given port.")
    # Send request
    send_all(s, request, True)
    # Receive reply
    reply = recv_all(s, timeout)
    # Close socket
    try:
        s.close()
    except socket.error:
        pass
    # Done
    return reply
if __name__ == "__main__":
    class Lala(RequestServer):
        def handle_request(self, req):
            print("REQ:", repr(req))
            return "The current time is %i" % time.time()
    s = Lala("localhost:test", 0, 1)
    s.start()
    if False:
        print(do_request("localhost:test", "wait", 5))
        for i in range(10):
            print(do_request("localhost:test", "hi" + str(i)))
    # do_request('localhost:test', 'wait'); do_request('localhost:test', 'hi');
```

```python
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
import os
import threading
import yoton
from yoton.misc import basestring
from yoton.misc import Property
# Minimum timout
TIMEOUT_MIN = 0.5
# For the status
STATUS_CLOSED = 0
STATUS_CLOSING = 1
STATUS_WAITING = 2
STATUS_HOSTING = 3
STATUS_CONNECTED = 4
STATUSMAP = [
    "closed",
    "closing",
    "waiting",
    "hosting",
    "connected",
]
# Reasons to stop the connection
STOP_DEFAULT_REASON = "Closed on command."
STOP_UNSPECIFIED_PROBLEM = "Unspecified problem"
STOP_INVALID_REASON = "Invalid stop reason specified (must be string)."
STOP_TIMEOUT = "Connection timed out."  # Can be used by user
STOP_HANDSHAKE_TIMEOUT = "Handshake timed out."
STOP_HANDSHAKE_FAILED = "Handshake failed."
STOP_HANDSHAKE_SELF = "Handshake failed (context cannot connect to self)."
STOP_CLOSED_FROM_THERE = "Closed from other end."
class ConnectionCollection(list):
    """ContextConnectionCollection()
    A list class that allows indexing using the name of the required
    Connection instance.
    """
    def __getitem__(self, key):
        if isinstance(key, basestring):
            if not key:
                raise KeyError("An empty string is not a valid key.")
            for c in self:
                if c.name == key:
                    return c
            else:
                raise KeyError("No connection know by the name %s" % key)
        else:
            return list.__getitem__(self, key)
class Connection(object):
    """Connection(context, name='')
    Abstract base class for a connection between two Context objects.
    This base class defines the full interface; subclasses only need
    to implement a few private methods.
    The connection classes are intended as a simple interface for the
    user, for example to query port number, and be notified of timeouts
    and closing of the connection.
    All connection instances are intended for one-time use. To make
    a new connection, instantiate a new Connection object. After
    instantiation, either _bind() or _connect() should be called.
    """
    def __init__(self, context, name=""):
        # Store context and name
        self._context = context
        self._name = name
        # Init hostname and port
        self._hostname1 = ""
        self._hostname2 = ""
        self._port1 = 0
        self._port2 = 0
        # Init id and pid of target context (set during handshake)
        # We can easily retrieve our own id and pid; no need to store
        self._id2 = 0
        self._pid2 = 0
        # Timeout value (if no data is received for this long,
        # the timedout signal is fired). Because we do not know the timeout
        # that the other side uses, we apply a minimum timeout.
        self._timeout = TIMEOUT_MIN
        # Create signals
        self._timedout_signal = yoton.Signal()
        self._closed_signal = yoton.Signal()
        # Lock to make setting and getting the status thread safe
        self._lock = threading.RLock()
        # Init variables to disconnected state
        self._set_status(0)
    ## Properties
    @property
    def hostname1(self):
        """Get the hostname corresponding to this end of the connection."""
        return self._hostname1
    @property
    def hostname2(self):
        """Get the hostname for the other end of this connection.
        Is empty string if not connected.
        """
        return self._hostname2
    @property
    def port1(self):
        """Get the port number corresponding to this end of the connection.
        When binding, use this port to connect the other context.
        """
        return self._port1
    @property
    def port2(self):
        """Get the port number for the other end of the connection.
        Is zero when not connected.
        """
        return self._port2
    @property
    def id1(self):
        """The id of the context on this side of the connection."""
        return self._context._id
    @property
    def id2(self):
        """The id of the context on the other side of the connection."""
        return self._id2
    @property
    def pid1(self):
        """The pid of the context on this side of the connection.
        (hint: os.getpid())
        """
        return os.getpid()
    @property
    def pid2(self):
        """The pid of the context on the other side of the connection."""
        return self._pid2
    @property
    def is_alive(self):
        """Get whether this connection instance is alive (i.e. either
        waiting or connected, and not in the process of closing).
        """
        self._lock.acquire()
        try:
            return self._status >= 2
        finally:
            self._lock.release()
    @property
    def is_connected(self):
        """Get whether this connection instance is connected."""
        self._lock.acquire()
        try:
            return self._status >= 3
        finally:
            self._lock.release()
    @property
    def is_waiting(self):
        """Get whether this connection instance is waiting for a connection.
        This is the state after using bind() and before another context
        connects to it.
        """
        self._lock.acquire()
        try:
            return self._status == 2
        finally:
            self._lock.release()
    @property
    def closed(self):
        """Signal emitted when the connection closes. The first argument
        is the ContextConnection instance, the second argument is the
        reason for the disconnection (as a string).
        """
        return self._closed_signal
    @Property
    def timeout():
        """Set/get the amount of seconds that no data is received from
        the other side after which the timedout signal is emitted.
        """
        def fget(self):
            return self._timeout
        def fset(self, value):
            if not isinstance(value, (int, float)):
                raise ValueError("timeout must be a number.")
            if value < TIMEOUT_MIN:
                raise ValueError("timeout must be at least %1.2f." %
TIMEOUT_MIN)
            self._timeout = value
        return locals()
    @property
    def timedout(self):
        """This signal is emitted when no data has been received for
        over 'timeout' seconds. This can mean that the connection is unstable,
        or that the other end is running extension code.
        Handlers are called with two arguments: the ContextConnection
        instance, and a boolean. The latter is True when the connection
        times out, and False when data is received again.
        """
        return self._timedout_signal
    @Property
    def name():
        """Set/get the name that this connection is known by. This name
        can be used to obtain the instance using the Context.connections
        property. The name can be used in networks in which each context
        has a particular role, to easier distinguish between the different
        connections. Other than that, the name has no function.
        """
        def fget(self):
            return self._name
        def fset(self, value):
            if not isinstance(value, basestring):
                raise ValueError("name must be a string.")
            self._name = value
        return locals()
    ## Public methods
    def flush(self, timeout=3.0):
        """flush(timeout=3.0)
        Wait until all pending packages are send. An error
        is raised when the timeout passes while doing so.
        """
        return self._flush(timeout)
    def close(self, reason=None):
        """close(reason=None)
        Close the connection, disconnecting the two contexts and
        stopping all trafic. If the connection was waiting for a
        connection, it stops waiting.
        Optionally, a reason for closing can be specified. A closed
        connection cannot be reused.
        """
        # No reason, user invoked close
        if reason is None:
            reason = STOP_DEFAULT_REASON
        # If already closed or closing, do nothing
        if self._status in [STATUS_CLOSED, STATUS_CLOSING]:
            return
        # Go!
        return self._general_close_method(reason, True)
    def close_on_problem(self, reason=None):
        """close_on_problem(reason=None)
        Disconnect the connection, stopping all trafic. If it was
        waiting for a connection, we stop waiting.
        Optionally, a reason for stopping can be specified. This is highly
        recommended in case the connection is closed due to a problem.
        In contrast to the normal close() method, this method does not
        try to notify the other end of the closing.
        """
        # No reason, some unspecified problem
        if reason is None:
            reason = STOP_UNSPECIFIED_PROBLEM
        # If already closed (status==0), do nothing
        if self._status == STATUS_CLOSED:
            return
        # If a connecion problem occurs during closing, we close the connection
        # so that flush will not block.
        # The closing that is now in progress will emit the event, so we
        # do not need to go into the _general_close_method().
        if self._status == STATUS_CLOSING:
            self._set_status(STATUS_CLOSED)
            return
        # Go!
        return self._general_close_method(reason, False)
    def _general_close_method(self, reason, send_stop_message):
        """general close method used by both close() and close_on_problem()"""
        # Remember current status. Set status to closing, which means that
        # the connection is still alive but cannot be closed again.
        old_status = self._status
        self._set_status(STATUS_CLOSING)
        # Check reason
        if not isinstance(reason, basestring):
            reason = STOP_INVALID_REASON
        # Tell other end to close?
        if send_stop_message and self.is_connected:
            self._notify_other_end_of_closing()
        # Close socket and set attributes to None
        self._set_status(STATUS_CLOSED)
        # Notify user, but only once
        self.closed.emit(self, reason)
        # Notify user ++
        if self._context._verbose:
            tmp = STATUSMAP[old_status]
            print("Yoton: %s connection closed: %s" % (tmp, reason))
    #         if True:
    #             tmp = STATUSMAP[old_status]
    #             sys.__stdout__.write("Yoton: %s connection closed: %s" % (tmp, 
reason))
    #             sys.__stdout__.flush()
    ## Methods to overload
    def _bind(self, hostname, port, max_tries):
        raise NotImplementedError()
    def _connect(self, hostname, port, timeout):
        raise NotImplementedError()
    def _flush(self, timeout):
        raise NotImplementedError()
    def _notify_other_end_of_closing(self):
        raise NotImplementedError()
    def _send_package(self, package):
        raise NotImplementedError()
    def _inject_package(self, package):
        raise NotImplementedError()
    def _set_status(self, status):
        """Used to change the status. Subclasses can reimplement this
        to get the desired behavior.
        """
        self._lock.acquire()
        try:
            # Store status
            self._status = status
            # Notify user ++
            if (status > 0) and self._context._verbose:
                action = STATUSMAP[status]
                print("Yoton: %s at %s:%s." % (action, self._hostname1, 
self._port1))
```

```
        finally:
            self._lock.release()
## More ideas for connections
class InterConnection(Connection):
    """InterConnection(context, hostname, port, name='')
    Not implemented.
    Communication between two processes on the same machine can also
    be implemented via a memory mapped file or a pype. Would there
    be an advantage over the TcpConnection?
    """
    pass
class UDPConnection(Connection):
    """UDPConnection(context, hostname, port, name='')
    Not implemented.
    Communication can also be done over UDP, but need protocol on top
    of UDP to make the connection robust. Is there a reason to implement
    this if we have Tcp?
    """
    pass
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
# flake8: noqa
import sys
import time
import yoton
from yoton.misc import basestring, bytes, str
from yoton.misc import Property, getErrorMsg, UID
from yoton.misc import PackageQueue
from yoton.connection import Connection, TIMEOUT_MIN
from yoton.connection import STATUS_CLOSED, STATUS_WAITING, STATUS_HOSTING
from yoton.connection import STATUS_CONNECTED, STATUS_CLOSING
class ItcConnection(Connection):
    """ItcConnection(context, hostname, port, name='')
    Not implemented .
    The inter-thread-communication connection class implements a
    connection between two contexts that are in the same process.
    Two instances of this class are connected using a weak reference.
    In case one of the ends is cleaned up by the garbadge collector,
    the other end will close the connection.
    """
    pass
    # todo: implement me
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
import os
import sys
import time
import socket
import threading
from yoton.misc import basestring, bytes, str
from yoton.misc import getErrorMsg, UID
from yoton.misc import TinyPackageQueue
from yoton.core import Package, PACKAGE_HEARTBEAT, PACKAGE_CLOSE, EINTR
from yoton.core import can_recv, send_all, recv_all, HEADER_SIZE
from yoton.connection import Connection, TIMEOUT_MIN  # noqa
from yoton.connection import STATUS_CLOSED, STATUS_WAITING, STATUS_HOSTING  #
noqa
from yoton.connection import STATUS_CONNECTED, STATUS_CLOSING  # noqa
# Note that time.sleep(0) yields the current thread's timeslice to any
# other >= priority thread in the process, but is otherwise equivalent 0 delay.
# Reasons to stop the connection
STOP_SOCKET_ERROR = "Socket error."  # the error message is appended
STOP_EOF = "Other end dropped the connection."
STOP_HANDSHAKE_TIMEOUT = "Handshake timed out."
STOP_HANDSHAKE_FAILED = "Handshake failed."
STOP_HANDSHAKE_SELF = "Handshake failed (context cannot connect to self)."
STOP_CLOSED_FROM_THERE = "Closed from other end."
STOP_LOST_TRACK = "Lost track of the stream."
STOP_THREAD_ERROR = "Error in io thread."
# Use a relatively small buffer size, to keep the channels better in sync
SOCKET_BUFFERS_SIZE = 10 * 1024
class TcpConnection(Connection):
    """TcpConnection(context, name='')
    The TcpConnection class implements a connection between two
    contexts that are in differenr processes or on different machines
    connected via the internet.
    This class handles the low-level communication for the context.
    A ContextConnection instance wraps a single BSD socket for its
    communication, and uses TCP/IP as the underlying communication
    protocol. A persisten connection is used (the BSD sockets stay
    connected). This allows to better distinguish between connection
    problems and timeouts caused by the other side being busy.
    """
    def __init__(self, context, name=""):
        # Variables to hold threads
        self._sendingThread = None
        self._receivingThread = None
        # Create queue for outgoing packages.
        self._qout = TinyPackageQueue(64, *context._queue_params)
        # Init normally (calls _set_status(0)
        Connection.__init__(self, context, name)
    def _set_status(self, status, bsd_socket=None):
        """_connected(status, bsd_socket=None)
        This method is called when a connection is made.
        Private method to apply the bsd_socket.
        Sets the socket and updates the status.
        Also instantiates the IO threads.
        """
        # Lock the connection while we change its status
        self._lock.acquire()
        # Update hostname and port number; for hosting connections the port
        # may be different if max_tries > 0. Each client connection will be
        # assigned a different ephemeral port number.
        # http://www.tcpipguide.com/free/t_TCPPortsConnectionsandConnectionIdent
ification-2.htm
        # Also get hostname and port for other end
        if bsd_socket is not None:
            if True:
                self._hostname1, self._port1 = bsd_socket.getsockname()
            if status != STATUS_WAITING:
                self._hostname2, self._port2 = bsd_socket.getpeername()
        # Set status as normal
        Connection._set_status(self, status)
        try:
            if status in [STATUS_HOSTING, STATUS_CONNECTED]:
                # Really connected
                # Store socket
                self._bsd_socket = bsd_socket
                # Set socket to blocking. Needed explicitly on Windows
                # One of these things it takes you hours to find out ...
                bsd_socket.setblocking(1)
                # Create and start io threads
                self._sendingThread = SendingThread(self)
                self._receivingThread = ReceivingThread(self)
                #
                self._sendingThread.start()
                self._receivingThread.start()
            if status == 0:
                # Close bsd socket
                try:
                    self._bsd_socket.shutdown()
                except Exception:
                    pass
                try:
                    self._bsd_socket.close()
                except Exception:
                    pass
                self._bsd_socket = None
                # Remove references to threads
                self._sendingThread = None
                self._receivingThread = None
        finally:
            self._lock.release()
    def _bind(self, hostname, port, max_tries=1):
        """Bind the bsd socket. Launches a dedicated thread that waits
        for incoming connections and to do the handshaking procedure.
        """
        # Create socket.
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        # Set buffer size to be fairly small (less than 10 packages)
        s.setsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF, SOCKET_BUFFERS_SIZE)
        s.setsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF, SOCKET_BUFFERS_SIZE)
        # Apply SO_REUSEADDR when binding, so that an improperly closed
        # socket on the same port will not prevent us from connecting.
        # It also allows a connection to bind at the same port number,
        # but only after the previous binding connection has connected
        # (and has closed the listen-socket).
        #
        # SO_REUSEADDR means something different on win32 than it does
        # for Linux sockets. To get the intended behavior on Windows,
        # we don't have to do anything. Also see:
        #  * http://msdn.microsoft.com/en-us/library/ms740621%28VS.85%29.aspx
        #  * http://twistedmatrix.com/trac/ticket/1151
        #  * http://www.tcpipguide.com/free/t_TCPPortsConnectionsandConnectionId
entification-2.htm
        if not sys.platform.startswith("win"):
            s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        # Try all ports in the specified range
        for try_count in range(1, max_tries + 1):
            port += int(try_count**0.5)
            try:
                s.bind((hostname, port))
                break
            except Exception:
                # Raise the socket exception if we were asked to try
                # just one port. Otherwise just try the next
                if max_tries == 1:
                    raise
                continue
        else:
            # We tried all ports without success
            tmp = "Could not bind to any of the %i %i ports tried."
            raise IOError(tmp % (max_tries, port))
        # Tell the socket it is a host. Backlog of at least 1 to avoid linux
        # kernel from detecting SYN flood and throttling the connection (#381)
        s.listen(1)
        # Set connected (status 1: waiting for connection)
        # Will be called with status 2 by the hostThread on success
        self._set_status(STATUS_WAITING, s)
        # Start thread to wait for a connection
        # (keep reference so the thread-object stays alive)
        self._hostThread = HostThread(self, s)
        self._hostThread.start()
    def _connect(self, hostname, port, timeout=1.0):
        """Connect to a bound socket."""
        # Create socket
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        # Set buffer size to be fairly small (less than 10 packages)
        s.setsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF, SOCKET_BUFFERS_SIZE)
        s.setsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF, SOCKET_BUFFERS_SIZE)
        # Refuse rediculously low timeouts
        if timeout <= 0.01:
            timeout = 0.01
        # Try to connect
        ok = False
        timestamp = time.time() + timeout
        while not ok and time.time() < timestamp:
            try:
                s.connect((hostname, port))
                ok = True
            except socket.error:
                pass
            except socket.timeout:
                pass
            time.sleep(timeout / 100.0)
        # Did it work?
        if not ok:
            type, value, tb = sys.exc_info()
            del tb
            err = str(value)
            raise IOError("Cannot connect to %s on %i: %s" % (hostname, port,
err))
        # Shake hands
        h = HandShaker(s)
        success, info = h.shake_hands_as_client(self.id1)
        # Problem?
        if not success:
            self._set_status(0)
            if not info:
                info = "problem during handshake"
            raise IOError("Could not connect: " + info)
        # Store id
        self._id2, self._pid2 = info
        # Set connected (status 3: connected as client)
        self._set_status(STATUS_CONNECTED, s)
    def _notify_other_end_of_closing(self):
        """Send PACKAGE_CLOSE."""
        self._qout.push(PACKAGE_CLOSE)
        try:
            self.flush(1.0)
        except Exception:
            pass  # Well, we tried ...
    def _flush(self, timeout=5.0):
        """Put a dummy message on the queue and spinlock until
        the thread has popped it from the queue.
        """
        # Check
        if not self.is_connected:
            RuntimeError("Cannot flush if not connected.")
        # Put dummy package on the queue
        self._qout.push(PACKAGE_HEARTBEAT)
        # Wait for the queue to empty. If it is, the heart beat package
        # may not been send yet, but every package befor it is.
        timestamp = time.time() + timeout
        while self.is_connected and not self._qout.empty():
            time.sleep(0.01)
            if time.time() > timestamp:
                raise RuntimeError("Sending the packages timed out.")
    def _send_package(self, package):
        """Put package on the queue, where the sending thread will
        pick it up.
        """
        self._qout.push(package)
    def _inject_package(self, package):
        """Put package in queue, but bypass potential blocking."""
        self._qout._q.append(package)
class HostThread(threading.Thread):
    """HostThread(context_connection, bds_socket)
    The host thread is used by the ContextConnection when hosting a
    connection. This thread waits for another context to connect
    to it, and then performs the handshaking procedure.
    When a successful connection is made, the context_connection's
    _connected() method is called and this thread then exits.
    """
    def __init__(self, context_connection, bsd_socket):
        threading.Thread.__init__(self)
        # Store connection and socket
        self._context_connection = context_connection
        self._bsd_host_socket = bsd_socket
        # Make deamon
        self.setDaemon(True)
    def run(self):
        """run()
        The main loop. Waits for a connection and performs handshaking
        if successfull.
        """
        # Try making a connection until success or the context is stopped
        while self._context_connection.is_waiting:
            # Wait for connection
            s = self._wait_for_connection()
            if not s:
                continue
            # Check if not closed in the mean time
            if not self._context_connection.is_waiting:
                break
            # Do handshaking
            hs = HandShaker(s)
            success, info = hs.shake_hands_as_host(self._context_connection.id1)
            if success:
                self._context_connection._id2 = info[0]
                self._context_connection._pid2 = info[1]
            else:
                print("Yoton: Handshake failed: " + info)
                continue
            # Success!
            # Close hosting socket, thereby enabling rebinding at the same port
            self._bsd_host_socket.close()
            # Update the status of the connection
            self._context_connection._set_status(STATUS_HOSTING, s)
            # Break out of the loop
            break
        # Remove ref
        del self._context_connection
        del self._bsd_host_socket
    def _wait_for_connection(self):
        """_wait_for_connection()
        The thread will wait here until someone connects. When a
        connections is made, the new socket is returned.
        """
        # Set timeout so that we can check _stop_me from time to time
        self._bsd_host_socket.settimeout(0.25)
        # Wait
        while self._context_connection.is_waiting:
            try:
                s, addr = self._bsd_host_socket.accept()
                return s  # Return the new socket
            except socket.timeout:
                pass
            except socket.error:
                # Skip errors caused by interruptions.
                type, value, tb = sys.exc_info()
                del tb
                if value.errno != EINTR:
                    raise
class HandShaker:
    """HandShaker(bsd_socket)
    Class that performs the handshaking procedure for Tcp connections.
    Essentially, the connecting side starts by sending 'YOTON!'
    followed by its id as a hex string. The hosting side responds
    with the same message (but with a different id).
    This process is very similar to a client/server pattern (both
    messages are also terminated with '\r\n'). This is done such that
    if for example a web client tries to connect, a sensible error
    message can be returned. Or when a ContextConnection tries to connect
    to a web server, it will be able to determine the error gracefully.
    """
    def __init__(self, bsd_socket):
        # Store bsd socket
        self._bsd_socket = bsd_socket
    def shake_hands_as_host(self, id):
        """_shake_hands_as_host(id)
        As the host, we wait for the client to ask stuff, so when
        for example a http client connects, we can stop the connection.
        Returns (success, info), where info is the id of the context at
        the other end, or the error message in case success is False.
        """
        # Make our message with id and pid
        message = "YOTON!%s.%i" % (UID(id).get_hex(), os.getpid())
        # Get request
        request = self._recv_during_handshaking()
        if not request:
            return False, STOP_HANDSHAKE_TIMEOUT
        elif request.startswith("YOTON!"):
            # Get id
            try:
                tmp = request[6:].split(".", 1)  # partition not in Python24
                id2_str, pid2_str = tmp[0], tmp[1]
                id2, pid2 = int(id2_str, 16), int(pid2_str, 10)
            except Exception:
                self._send_during_handshaking("ERROR: could not parse id.")
                return False, STOP_HANDSHAKE_FAILED
            # Respond and return
            self._send_during_handshaking(message)  # returns error?
            if id == id2:
                return False, STOP_HANDSHAKE_SELF
            else:
                return True, (id2, pid2)
        else:
            # Client is not yoton
            self._send_during_handshaking("ERROR: this is Yoton.")
            return False, STOP_HANDSHAKE_FAILED
    def shake_hands_as_client(self, id):
        """_shake_hands_as_client(id)
        As the client, we ask the host whether it is a Yoton context
        and whether the channels we want to support are all right.
        Returns (success, info), where info is the id of the context at
        the other end, or the error message in case success is False.
        """
        # Make our message with id and pif
        message = "YOTON!%s.%i" % (UID(id).get_hex(), os.getpid())
        # Do request
        self._send_during_handshaking(message)  # returns error?
        # Get response
        response = self._recv_during_handshaking()
        # Process
        if not response:
            return False, STOP_HANDSHAKE_TIMEOUT
        elif response.startswith("YOTON!"):
            # Get id
            try:
                tmp = response[6:].split(".", 1)  # Partition not in Python24
                id2_str, pid2_str = tmp[0], tmp[1]
                id2, pid2 = int(id2_str, 16), int(pid2_str, 10)
            except Exception:
                return False, STOP_HANDSHAKE_FAILED
            if id == id2:
                return False, STOP_HANDSHAKE_SELF
            else:
                return True, (id2, pid2)
        else:
            return False, STOP_HANDSHAKE_FAILED
    def _send_during_handshaking(self, text, shutdown=False):
        return send_all(self._bsd_socket, text + "\r\n", shutdown)
    def _recv_during_handshaking(self):
        return recv_all(self._bsd_socket, 2.0, True)
class BaseIOThread(threading.Thread):
    """The base class for the sending and receiving IO threads.
    Implements some common functionality.
    """
    def __init__(self, context_connection):
        threading.Thread.__init__(self)
        # Thread will "destruct" when the interpreter shuts down
        self.setDaemon(True)
        # Store (temporarily) the ref to the context connection
        # Also of bsd_socket, because it might be removed before the
        # thread is well up and running.
        self._context_connection = context_connection
        self._bsd_socket = context_connection._bsd_socket
    def run(self):
        """Method to prepare to enter main loop. There is a try-except here
        to catch exceptions caused by interpreter shutdown.
        """
        # Get ref to context connection but make sure the ref
        # if not stored if the thread stops
        context_connection = self._context_connection
        bsd_socket = self._bsd_socket
        del self._context_connection
        del self._bsd_socket
        try:
            # Run and handle exceptions if someting goes wrong
            self.run2(context_connection, bsd_socket)
        except Exception:
            # An error while handling an exception, most probably
            # interpreter shutdown
            pass
    def run2(self, context_connection, bsd_socket):
        """Method to enter main loop. There is a try-except here to
        catch exceptions in the main loop (such as socket errors and
        errors due to bugs in the code.
        """
        # Get classname to use in messages
        className = "yoton." + self.__class__.__name__
        # Define function to write errors
        def writeErr(err):
            sys.__stderr__.write(str(err) + "\n")
            sys.__stderr__.flush()
        try:
            # Enter mainloop
            stop_reason = self._run(context_connection, bsd_socket)
            # Was there a specific reason to stop?
            if stop_reason:
                context_connection.close_on_problem(stop_reason)
            else:
                pass  # Stopped because the connection is gone (already stopped)
        except socket.error:
            # Socket error. Can happen if the other end closed not so nice
            # Do get the socket error message and pass it on.
            msg = STOP_SOCKET_ERROR + getErrorMsg()
            context_connection.close_on_problem("%s, %s" % (className, msg))
        except Exception:
            # Else: woops, an error!
            errmsg = getErrorMsg()
            msg = STOP_THREAD_ERROR + errmsg
            context_connection.close_on_problem("%s, %s" % (className, msg))
            writeErr("Exception in %s." % className)
            writeErr(errmsg)
class SendingThread(BaseIOThread):
    """The thread that reads packages from the queue and sends them over
    the socket. It uses a timeout while reading from the queue, so it can
    send heart beat packages if no packages are send.
    """
    def _run(self, context_connection, bsd_socket):
        """The main loop. Get package from queue, send package to socket."""
        timeout = 0.5 * TIMEOUT_MIN
        queue = context_connection._qout
        socket_sendall = bsd_socket.sendall
        while True:
            time.sleep(0)  # Be nice
            # Get package from the queue. Use heartbeat package
            # if there have been no packages for a too long time
            try:
                package = queue.pop(timeout)
            except queue.Empty:
                # Use heartbeat package then
                package = PACKAGE_HEARTBEAT
                # Should we stop?
                if not context_connection.is_connected:
                    return None  # No need for a stop message
            # Process the package in parts to avoid data copying (header+data)
            for part in package.parts():
                socket_sendall(part)
class ReceivingThread(BaseIOThread):
    """The thread that reads packages from the socket and passes them to
    the kernel. It uses select() to see if data is available on the socket.
    This allows using a timeout without putting the socket in timeout mode.
    If the timeout has expired, the timedout event for the connection is
    emitted.
    Upon receiving a package, the _recv_package() method of the context
    is called, so this thread will eventually dispose the package in
    one or more queues (of the channel or of another connection).
    """
    def _run(self, context_connection, bsd_socket):
        """The main loop. Get package from socket, deposit package in
queue(s)."""
        # Short names in local namespace avoid dictionary lookups
        socket_recv = bsd_socket.recv
        recv_package = context_connection._context._recv_package
        package_from_header = Package.from_header
        HS = HEADER_SIZE
        # Variable to keep track if we emitted a timedout signal
        timedOut = False
        while True:
            time.sleep(0)  # Be nice
            # Use select call on a timeout to test if data is available
            while True:
                try:
                    ok = can_recv(bsd_socket, context_connection._timeout)
                except Exception:
                    # select() has it's share of weird errors
                    raise socket.error("select(): " + getErrorMsg())
                if ok:
                    # Set timedout ex?
                    if timedOut:
                        timedOut = False
                        context_connection.timedout.emit(context_connection,
False)
                    # Exit from loop
                    break
                else:
                    # Should we stop?
                    if not context_connection.is_connected:
                        return  # No need for a stop message
                    # Should we do a timeout?
                    if not timedOut:
                        timedOut = True
                        context_connection.timedout.emit(context_connection,
True)
                    # Continue in loop
                    continue
            # Get package
            package = self._getPackage(socket_recv, HS, package_from_header)
            if package is None:
                continue
            elif isinstance(package, basestring):
                return package  # error msg
            # Let context handle package further (but happens in this thread)
            try:
                recv_package(package, context_connection)
            except Exception:
                print("Error depositing package in ReceivingThread.")
                print(getErrorMsg())
    def _getPackage(self, socket_recv, HS, package_from_header):
        """Get exactly one package from the socket. Blocking."""
        # Get header and instantiate package object from it
        try:
            header = self._recv_n_bytes(socket_recv, HS)
        except EOFError:
            return STOP_EOF
        package, size = package_from_header(header)
        # Does it look like a good package?
        if not package:
            return STOP_LOST_TRACK
        if size == 0:
            # A special package! (or someone sending a
            # package with no content, which is discarted)
            if package._source_seq == 0:
                pass  # Heart beat
            elif package._source_seq == 1:
                return STOP_CLOSED_FROM_THERE
            return None
        else:
            # Get package data
            try:
                package._data = self._recv_n_bytes(socket_recv, size)
            except EOFError:
                return STOP_EOF
            return package
    def _recv_n_bytes(self, socket_recv, n):
        """Receive exactly n bytes from the socket."""
        # First round
        data = socket_recv(n)
        if len(data) == 0:
            raise EOFError()
        # How many more do we need? For small n, we probably only need 1 round
        n -= len(data)
        if n == 0:
            return data  # We're lucky!
        # Else, we need more than one round
```

```
parts = [data]
while n:
    data = socket_recv(n)
    parts.append(data)
    n -= len(data)
# Return combined data of multiple rounds
return bytes().join(parts)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module yoton.context
Defines the context class.
"""
import threading
from yoton import connection
from yoton.misc import str, split_address
from yoton.misc import UID, PackageQueue
from yoton.core import Package, BUF_MAX_LEN
from yoton.core import SLOT_CONTEXT
from yoton.connection import ConnectionCollection
from yoton.connection_tcp import TcpConnection
from yoton.connection_itc import ItcConnection
class Context(object):
    """Context(verbose=0, queue_params=None)
    A context represents a node in the network. It can connect to
    multiple other contexts (using a yoton.Connection.
    These other contexts can be in
    another process on the same machine, or on another machine
    connected via a network or the internet.
    This class represents a context that can be used by channel instances
    to communicate to other channels in the network. (Thus the name.)
    The context is the entity that queue routes the packages produced
    by the channels to the other context in the network, where
    the packages are distributed to the right channels. A context queues
    packages while it is not connected to any other context.
    If messages are send on a channel registered at this context while
    the context is not connected, the messages are stored by the
    context and will be send to the first connecting context.
    Example 1
    ---------
    # Create context and bind to a port on localhost
    context = yoton.Context()
    context.bind('localhost:11111')
    # Create a channel and send a message
    pub = yoton.PubChannel(context, 'test')
    pub.send('Hello world!')
    Example 2
    ---------
    # Create context and connect to the port on localhost
    context = yoton.Context()
    context.connect('localhost:11111')
    # Create a channel and receive a message
    sub = yoton.SubChannel(context, 'test')
    print(sub.recv() # Will print 'Hello world!'
    Queue params
    ------------
    The queue_params parameter allows one to specify the package queues
    used in the system. It is recommended to use the same parameters
    for every context in the network. The value of queue_params should
    be a 2-element tuple specifying queue size and discard mode. The
    latter can be 'old' (default) or 'new', meaning that if the queue
    is full, either the oldest or newest messages are discarted.
    """
    def __init__(self, verbose=0, queue_params=None):
        # Whether or not to write status information
        self._verbose = verbose
        # Store queue parameters
        if queue_params is None:
            queue_params = BUF_MAX_LEN, "old"
        if not (isinstance(queue_params, tuple) and len(queue_params) == 2):
            raise ValueError("queue_params should be a 2-element tuple.")
        self._queue_params = queue_params
        # Create unique key to identify this context
        self._id = UID().get_int()
        # Channels currently registered. Maps slots to channel instance.
        self._sending_channels = {}
        self._receiving_channels = {}
        # The list of connections
        self._connections = []
        self._connections_lock = threading.RLock()
        # Queue used during startup to collect packages
        # This queue is also subject to the _connections_lock
        self._startupQueue = PackageQueue(*queue_params)
        # For numbering and routing the packages
        self._send_seq = 0
        self._recv_seq = 0
        self._source_map = {}
    def close(self):
        """close()
        Close the context in a nice way, by closing all connections
        and all channels.
        Closing a connection means disconnecting two contexts. Closing
        a channel means disasociating a channel from its context.
        Unlike connections and channels, a Context instance can be reused
        after closing (although this might not always the best strategy).
        """
        # Close all connections (also the waiting connections!)
        for c in self.connections_all:
            c.close("Closed by the context.")
        # Close all channels
        self.close_channels()
    def close_channels(self):
        """close_channels()
        Close all channels associated with this context. This does
        not close the connections. See also close().
        """
        # Get all channels
        channels1 = [c for c in self._sending_channels.values()]
        channels2 = [c for c in self._receiving_channels.values()]
        # Close all channels
        for c in set(channels1 + channels2):
            c.close()
    ## Properties
    @property
    def connections_all(self):
        """Get a list of all Connection instances currently
        associated with this context, including pending connections
        (connections waiting for another end to connect).
        In addition to normal list indexing, the connections objects can be
        queried from this list using their name.
        """
        # Lock
        self._connections_lock.acquire()
        try:
            return [c for c in self._connections if c.is_alive]
        finally:
            self._connections_lock.release()
    @property
    def connections(self):
        """Get a list of the Connection instances currently
        active for this context.
        In addition to normal list indexing, the connections objects can be
        queried  from this list using their name.
        """
        # Lock
        self._connections_lock.acquire()
        try:
            # Clean up any dead connections
            copy = ConnectionCollection()
            to_remove = []
            for c in self._connections:
                if not c.is_alive:
                    to_remove.append(c)
                elif c.is_connected:
                    copy.append(c)
            # Clean
            for c in to_remove:
                self._connections.remove(c)
            # Return copy
            return copy
        finally:
            self._connections_lock.release()
    @property
    def connection_count(self):
        """Get the number of connected contexts. Can be used as a boolean
        to check if the context is connected to any other context.
        """
        return len(self.connections)
    @property
    def id(self):
        """The 8-byte UID of this context."""
        return self._id
    ## Public methods
    def bind(self, address, max_tries=1, name=""):
        """bind(address, max_tries=1, name='')
        Setup a connection with another Context, by being the host.
        This method starts a thread that waits for incoming connections.
        Error messages are printed when an attempted connect fails. the
        thread keeps trying until a successful connection is made, or until
        the connection is closed.
        Returns a Connection instance that represents the
        connection to the other context. These connection objects
        can also be obtained via the Context.connections property.
        Parameters
        ----------
        address : str
            Should be of the shape hostname:port. The port should be an
            integer number between 1024 and 2**16. If port does not
            represent a number, a valid port number is created using a
            hash function.
        max_tries : int
            The number of ports to try; starting from the given port,
            subsequent ports are tried until a free port is available.
            The final port can be obtained using the 'port' property of
            the returned Connection instance.
        name : string
            The name for the created Connection instance. It can
            be used as a key in the connections property.
        Notes on hostname
        -----------------
        The hostname can be:
          * The IP address, or the string hostname of this computer.
          * 'localhost': the connections is only visible from this computer.
            Also some low level networking layers are bypassed, which results
            in a faster connection. The other context should also connect to
            'localhost'.
          * 'publichost': the connection is visible by other computers on the
            same network. Optionally an integer index can be appended if
            the machine has multiple IP addresses (see socket.gethostbyname_ex).
        """
        # Trigger cleanup of closed connections
        self.connections
        # Split address in protocol, real hostname and port number
        protocol, hostname, port = split_address(address)
        # Based on protocol, instantiate connection class (currently only tcp)
        if False:  # protocol == 'itc':
            connection = ItcConnection(self, name)
        else:
            connection = TcpConnection(self, name)
        # Bind connection
        connection._bind(hostname, port, max_tries)
        # Save connection instance
        self._connections_lock.acquire()
        try:
            # Push packages from startup queue
            while len(self._startupQueue):
                connection._inject_package(self._startupQueue.pop())
            # Add connection object to list of connections
            self._connections.append(connection)
        finally:
            self._connections_lock.release()
        # Return Connection instance
        return connection
    def connect(self, address, timeout=1.0, name=""):
        """connect(self, address, timeout=1.0, name='')
        Setup a connection with another context, by connection to a
        hosting context. An error is raised when the connection could
        not be made.
        Returns a Connection instance that represents the
        connection to the other context. These connection objects
        can also be obtained via the Context.connections property.
        Parameters
        ----------
        address : str
            Should be of the shape hostname:port. The port should be an
            integer number between 1024 and 2**16. If port does not
            represent a number, a valid port number is created using a
            hash function.
        max_tries : int
            The number of ports to try; starting from the given port,
            subsequent ports are tried until a free port is available.
            The final port can be obtained using the 'port' property of
            the returned Connection instance.
        name : string
            The name for the created Connection instance. It can
            be used as a key in the connections property.
        Notes on hostname
        -----------------
        The hostname can be:
          * The IP address, or the string hostname of this computer.
          * 'localhost': the connection is only visible from this computer.
            Also some low level networking layers are bypassed, which results
            in a faster connection. The other context should also host as
            'localhost'.
          * 'publichost': the connection is visible by other computers on the
            same network. Optionally an integer index can be appended if
            the machine has multiple IP addresses (see socket.gethostbyname_ex).
        """
        # Trigger cleanup of closed connections
        self.connections
        # Split address in protocol, real hostname and port number
        protocol, hostname, port = split_address(address)
        # Based on protocol, instantiate connection class (currently only tcp)
        if False:  # protocol == 'itc':
            connection = ItcConnection(self, name)
        else:
            connection = TcpConnection(self, name)
        # Create new connection and connect it
        connection._connect(hostname, port, timeout)
        # Save connection instance
        self._connections_lock.acquire()
        try:
            # Push packages from startup queue
            while self._startupQueue:
                connection._inject_package(self._startupQueue.pop())
            # Add connection object to list of connections
            self._connections.append(connection)
        finally:
            self._connections_lock.release()
        # Send message in the network to signal a new connection
        bb = "NEW_CONNECTION".encode("utf-8")
        p = Package(bb, SLOT_CONTEXT, self._id, 0, 0, 0, 0)
        self._send_package(p)
        # Return Connection instance
        return connection
    def flush(self, timeout=5.0):
        """flush(timeout=5.0)
        Wait until all pending messages are send. This will flush all
        messages posted from the calling thread. However, it is not
        guaranteed that no new messages are posted from another thread.
        Raises an error when the flushing times out.
        """
        # Flush all connections
        for c in self.connections:
            c.flush(timeout)
        # Done (backward compatibility)
        return True
    ## Private methods used by the Channel classes
    def _register_sending_channel(self, channel, slot, slotname=""):
        """_register_sending_channel(channel, slot, slotname='')
        The channel objects use this method to register themselves
        at a particular slot.
        """
        # Check if this slot is free
        if slot in self._sending_channels:
            raise ValueError("Slot not free: " + str(slotname))
        # Register
        self._sending_channels[slot] = channel
    def _register_receiving_channel(self, channel, slot, slotname=""):
        """_register_receiving_channel(channel, slot, slotname='')
        The channel objects use this method to register themselves
        at a particular slot.
        """
        # Check if this slot is free
        if slot in self._receiving_channels:
            raise ValueError("Slot not free: " + str(slotname))
        # Register
        self._receiving_channels[slot] = channel
    def _unregister_channel(self, channel):
        """_unregister_channel(channel)
        Unregisters the given channel. That channel can no longer
        receive messages, and should no longer send messages.
        """
        for D in [self._receiving_channels, self._sending_channels]:
            for key in [key for key in D.keys()]:
                if D[key] == channel:
                    D.pop(key)
    ## Private methods to pass packages between context and io-threads
    def _send_package(self, package):
        """_send_package(package)
        Used by the channels to send a package into the network.
        This method routes the package to all currentlt connected
        connections. If there are none, the packages is queued at
        the context.
        """
        # Add number
        self._send_seq += 1
        package._source_seq = self._send_seq
        # Send to all connections, or queue if there are none
        self._connections_lock.acquire()
        try:
            ok = False
            for c in self._connections:
                if c.is_alive:  # Waiting or connected
                    c._send_package(package)
                    ok = True
            # Should we queue the package?
            if not ok:
                self._startupQueue.push(package)
        finally:
            self._connections_lock.release()
    def _recv_package(self, package, connection):
        """_recv_package(package, connection)
        Used by the connections to receive a package at this
        context. The package is distributed to all connections
        except the calling one. The package is also distributed
        to the right channel (if applicable).
        """
        # Get slot
        slot = package._slot
        # Init what to do with the package
        send_further = False
        deposit_here = False
        # Get what to do with the package
        last_seq = self._source_map.get(package._source_id, 0)
        if last_seq < package._source_seq:
            # Update source map
            self._source_map[package._source_id] = package._source_seq
            if package._dest_id == 0:
                # Add to both lists, first attach seq nr
                self._recv_seq += 1
                package._recv_seq = self._recv_seq
                send_further, deposit_here = True, True
            elif package._dest_id == self._id:
                # Add only to process list, first attach seq nr
                self._recv_seq += 1
                package._recv_seq = self._recv_seq
                deposit_here = True
            else:
                # Send package to connected nodes
                send_further = True
        # Send package to other context (over all alive connections)
        if send_further:
            self._connections_lock.acquire()
            try:
                for c in self._connections:
                    if c is connection or not c.is_alive:
                        continue
                    c._send_package(package)
            finally:
                self._connections_lock.release()
        # Process package here or pass to channel
        if deposit_here:
            if slot == SLOT_CONTEXT:
                # Context-to-context messaging;
                # A slot starting with a space reprsents the context
                self._recv_context_package(package)
            else:
                # Give package to a channel (if applicable)
                channel = self._receiving_channels.get(slot, None)
                if channel is not None:
                    channel._recv_package(package)
    def _recv_context_package(self, package):
        """_recv_context_package(package)
        Process a package addressed at the context itself. This is how
        the context handles higher-level connection tasks.
        """
        # Get message: context messages are always utf-8 encoded strings
        message = package._data.decode("utf-8")
        if message == "CLOSE_CONNECTION":
            # Close the connection. Check which one of our connections is
            # connected with the context that send this message.
            self._connections_lock.acquire()
            try:
                for c in self.connections:
                    if c.is_connected and c.id2 == package._source_id:
                        c.close(connection.STOP_CLOSED_FROM_THERE, False)
            finally:
                self._connections_lock.release()
        elif message == "NEW_CONNECTION":
            # Resend all status channels
            for channel in self._sending_channels.values():
                if hasattr(channel, "_current_message") and hasattr(
                    channel, "send_last"
                ):
                    channel.send_last()
        else:
            print("Yoton: Received unknown context message: " + message)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
import sys
import time
import struct
import socket
# import select  # to determine wheter a socket can receive data
if sys.platform.startswith("java"):  # Jython
    from select import cpython_compatible_select as select, error as SelectErr
else:
    from select import select, error as SelectErr
from yoton.misc import bytes
## Constants
# Error code for interruptions
try:
    from errno import EINTR
except ImportError:
    EINTR = None
# Queue size
BUF_MAX_LEN = 10000
# Define buffer size. For recv 4096 or 8192 chunk size is recommended.
# For sending, in principle as large as possible, but prevent too much
# message copying.
BUFFER_SIZE_IN = 2**13
BUFFER_SIZE_OUT = 1 * 2**20
# Reserved slots (slot 0 does not exist, so we can test "if slot: ")
# The actual slots start counting from 8.
# SLOT_TEST = 3 -> Deprecated
SLOT_CONTEXT = 2
SLOT_DUMMY = 1
# Struct header packing
HEADER_FORMAT = "<QQQQQQQ"
HEADER_SIZE = struct.calcsize(HEADER_FORMAT)
# Constant for control bytes
CONTROL_BYTES = struct.unpack("<Q", "YOTON   ".encode("utf-8"))[0]
## Helper functions
def can_send(s, timeout=0.0):
    """can_send(bsd_socket, timeout=0.0)
    Given a socket, uses select() to determine whether it can be used
    for sending. This function can deal with system interrupts.
    """
    while True:
        try:
            can_recv, can_send, tmp = select([], [s], [], timeout)
            break
        except SelectErr:
            # Skip errors caused by interruptions.
            type, value, tb = sys.exc_info()
            del tb
            if value.args[0] != EINTR:
                raise
    return bool(can_send)
def can_recv(s, timeout=0.0):
    """can_recv(bsd_socket, timeout=0.0)
    Given a socket, uses select() to determine whether it can be used
    for receiving. This function can deal with system interrupts.
    """
    while True:
        try:
            can_recv, can_send, tmp = select([s], [], [], timeout)
            break
        except SelectErr:
            # Skip errors caused by interruptions.
            type, value, tb = sys.exc_info()
            del tb
            if value.args[0] != EINTR:
                raise
    return bool(can_recv)
def send_all(s, text, stutdown_after_sending=True):
    """send_all(socket, text, stutdown_after_sending=True)
    Send all text to the socket. Used during handshaking and in
    the clientserver module.
    If stutdown_after_sending, the socket is shut down. Some protocols
    rely on this.
    It is made sure that the text ends with a CRLF double-newline code.
    """
    # Ensure closing chars
    if not text.endswith("\r\n"):
        text += "\r\n"
    # Make bytes
    bb = text.encode("utf-8")
    # Send all bytes
    try:
        s.sendall(bb)  # -> n
    except socket.error:
        return -1  # Socket closed down badly
    # Shutdown connection nicely from here
    if stutdown_after_sending:
        try:
            s.shutdown(socket.SHUT_WR)
        except socket.error:
            pass
def recv_all(s, timeout=-1, end_at_crlf=True):
    """recv_all(socket, timeout=-1, end_at_crlf=True)
    Receive text from the socket (untill socket receiving is shut down).
    Used during handshaking and in the clientserver module.
    If end_at_crlf, a message is also ended at a CRLF double-newline code,
    and a shutdown is not necessary. This takes a tiny bit longer.
    """
    # Init parts (start with one byte, such that len(parts) is always >= 2
    parts = [
        " ".encode("ascii"),
    ]
    # Determine number of bytes to get per recv
    nbytesToGet = BUFFER_SIZE_IN
    if end_at_crlf:
        nbytesToGet = 1
    # Set end bytes
    end_bytes = "\r\n".encode("ascii")
    # Set max time
    if timeout <= 0:
        timeout = 2**32
    maxtime = time.time() + timeout
    # Receive data
    while True:
        # Receive if we can
        if can_recv(s):
            # Get part
            try:
                part = s.recv(nbytesToGet)
                parts.append(part)
            except socket.error:
                return None  # Socket closed down badly
            # Detect end by shutdown (EOF)
            if not part:
                break
            # Detect end by \r\n
            if end_at_crlf and (parts[-2] + parts[-1]).endswith(end_bytes):
                break
        else:
            # Sleep
            time.sleep(0.01)
            # Check time
            if time.time() > maxtime:
                bb = bytes().join(parts[1:])
                return bb.decode("utf-8", "ignore")
    # Combine parts (discared first (dummy) part)
    bb = bytes().join(parts[1:])
    # Try returning as Unicode
    try:
        return bb.decode("utf-8", "ignore")
    except UnicodeError:
        return "<UnicodeError>"
## Package class
class Package(object):
    """Package(data, slot, source_id, source_seq, dest_id, dest_seq, recv_seq)
    Represents a package of bytes to be send from one Context instance
    to another. A package consists of a header and the encoded message.
    To make this class as fast as reasonably possible, its interface
    is rather minimalistic and few convenience stuff is implemented.
    Parameters
    ----------
    data : bytes
        The message itself.
    slot : long
        The slot at which the package is directed. The integer is a hash of
        a string slot name.
    source_id : long
        The id of the context that sent this package.
    source_seq : long
        The sequence number of this package, counted at the sending context.
        Together with source_id, this fully identifies a package.
    dest_id : long (default 0)
        The id of the package that this package replies to.
    dest_seq : long (default 0)
        The sequence number of the package that this package replies to.
    recv_seq : long (default 0)
        The sequence number of this package counted at the receiving context.
        This is used to synchronize channels.
    When send, the header is composed of four control bytes, the slot,
    the source_id, source_seq, dest_id and dest_seq.
    Notes
    -----
    A package should always have content. Packages without content are only
    used for low-level communication between two ContextConnection instances.
    The source_seq is then used as the signal. All other package attributes
    are ignored.
    """
    # The __slots__ makes instances of this class consume < 20% of memory
    # Note that this only works for new style classes.
    # This is important because many packages can exist at the same time
    # if a receiver cant keep up with a sender. Further, although Python's
    # garbage collector collects the objects after they're "consumed",
    # it does not release the memory, because it hopes to reuse it in
    # an efficient way later.
    __slots__ = [
        "_data",
        "_slot",
        "_source_id",
        "_source_seq",
        "_dest_id",
        "_dest_seq",
        "_recv_seq",
    ]
    def __init__(self, data, slot, source_id, source_seq, dest_id, dest_seq,
recv_seq):
        self._data = data
        self._slot = slot
        #
        self._source_id = source_id
        self._source_seq = source_seq
        self._dest_id = dest_id
        self._dest_seq = dest_seq
        self._recv_seq = recv_seq
    def parts(self):
        """parts()
        Get list of bytes that represents this package.
        By not concatenating the header and content parts,
        we prevent unnecesary copying of data.
        """
        # Obtain header
        L = len(self._data)
        header = struct.pack(
            HEADER_FORMAT,
            CONTROL_BYTES,
            self._slot,
            self._source_id,
            self._source_seq,
            self._dest_id,
            self._dest_seq,
            L,
        )
        # Return header and message
        return header, self._data
    def __str__(self):
        """Representation of the package. Mainly for debugging."""
        return "At slot %i: %s" % (self._slot, repr(self._data))
    @classmethod
    def from_header(cls, header):
        """from_header(header)
        Create a package (without data) from the header of a message.
        Returns (package, data_length). If the header is invalid (checked
        using the four control bytes) this method returns (None, None).
        """
        # Unpack
        tmp = struct.unpack(HEADER_FORMAT, header)
        CTRL, slot, source_id, source_seq, dest_id, dest_seq, L = tmp
        # Create package
        p = Package(None, slot, source_id, source_seq, dest_id, dest_seq, 0)
        # Return
        if CTRL == CONTROL_BYTES:
            return p, L
        else:
            return None, None
# Constant Package instances (source_seq represents the signal)
PACKAGE_HEARTBEAT = Package(bytes(), SLOT_DUMMY, 0, 0, 0, 0, 0)
PACKAGE_CLOSE = Package(bytes(), SLOT_DUMMY, 0, 1, 0, 0, 0)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
# This code is loosely based on the event system of Visvis and on the
# signals system of Qt.
# Note: Python has a buildin module (sched) that does some of the things
# here. Hoever, only since Python3.3 is this buildin functionality
# thread safe. And we need thread safety!
""" Module yoton.events
Yoton comes with a simple event system to enable event-driven applications.
All channels are capable of running without the event system, but some
channels have limitations. See the documentation of the channels for
more information. Note that signals only work if events are processed.
"""
import time
import threading
import weakref
from yoton.misc import Property, getErrorMsg, PackageQueue
class CallableObject(object):
    """CallableObject(callable)
    A class to hold a callable. If it is a plain function, its reference
    is held (because it might be a closure). If it is a method, we keep
    the function name and a weak reference to the object. In this way,
    having for instance a signal bound to a method, the object is not
    prevented from being cleaned up.
    """
    __slots__ = ["_ob", "_func"]  # Use __slots__ to reduce memory footprint
    def __init__(self, c):
        # Check
        if not hasattr(c, "__call__"):
            raise ValueError("Error: given callback is not callable.")
        # Store funcion and object
        if hasattr(c, "__self__"):
            # Method, store object and method name
            self._ob = weakref.ref(c.__self__)
            self._func = c.__func__.__name__
        elif hasattr(c, "im_self"):
            # Method in older Python
            self._ob = weakref.ref(c.im_self)
            self._func = c.im_func.__name__
        else:
            # Plain function
            self._func = c
            self._ob = None
    def isdead(self):
        """Get whether the weak ref is dead."""
        if self._ob:
            # Method
            return self._ob() is None
        else:
            return False
    def compare(self, other):
        """compare this instance with another."""
        if self._ob and other._ob:
            return (self._ob() is other._ob()) and (self._func == other._func)
        elif not (self._ob or other._ob):
            return self._func == other._func
        else:
            return False
    def __str__(self):
        return self._func.__str__()
    def call(self, *args, **kwargs):
        """call(*args, **kwargs)
        Call the callable. Exceptions are caught and printed.
        """
        if self.isdead():
            return
        # Get function
        try:
            if self._ob:
                func = getattr(self._ob(), self._func)
            else:
                func = self._func
        except Exception:
            return
        # Call it
        try:
            return func(*args, **kwargs)
        except Exception:
            print("Exception while handling event:")
            print(getErrorMsg())
class Event(object):
    """Event(callable, *args, **kwargs)
    An Event instance represents something that is going to be done.
    It consists of a callable and arguments to call it with.
    Instances of this class populate the event queue.
    """
    __slots__ = ["_callable", "_args", "_kwargs", "_timeout"]
    def __init__(self, callable, *args, **kwargs):
        if isinstance(callable, CallableObject):
            self._callable = callable
        else:
            self._callable = CallableObject(callable)
        self._args = args
        self._kwargs = kwargs
    def dispatch(self):
        """dispatch()
        Call the callable with the arguments and keyword-arguments specified
        at initialization.
        """
        self._callable.call(*self._args, **self._kwargs)
    def _on_timeout(self):
        """This is what theTimerThread calls."""
        app.post_event(self)
class Signal:
    """Signal()
    The purpose of a signal is to provide an interface to bind/unbind
    to events and to fire them.
    One can bind() or unbind() a callable to the signal. When emitted, an
    event is created for each bound handler. Therefore, the event loop
    must run for signals to work.
    Some signals call the handlers using additional arguments to
    specify specific information.
    """
    def __init__(self):
        self._handlers = []
    @property
    def type(self):
        """The type (__class__) of this event."""
        return self.__class__
    def bind(self, func):
        """bind(func)
        Add an eventhandler to this event.
        The callback/handler (func) must be a callable. It is called
        with one argument: the event instance, which can contain
        additional information about the event.
        """
        # make callable object (checks whether func is callable)
        cnew = CallableObject(func)
        # check -> warn
        for c in self._handlers:
            if cnew.compare(c):
                print("Warning: handler %s already present for %s" % (func, 
self))
                return
        # add the handler
        self._handlers.append(cnew)
    def unbind(self, func=None):
        """unbind(func=None)
        Unsubscribe a handler, If func is None, remove all handlers.
        """
        if func is None:
            self._handlers[:] = []
        else:
            cref = CallableObject(func)
            for c in [c for c in self._handlers]:
                # remove if callable matches func or object is destroyed
                if c.compare(cref) or c.isdead():
                    self._handlers.remove(c)
    def emit(self, *args, **kwargs):
        """emit(*args, **kwargs)
        Emit the signal, calling all bound callbacks with *args and **kwargs.
        An event is queues for each callback registered to this signal.
        Therefore it is safe to call this method from another thread.
        """
        # Add an event for each callback
        toremove = []
        for func in self._handlers:
            if func.isdead():
                toremove.append(func)
            else:
                event = Event(func, *args, **kwargs)
                app.post_event(event)
        # Remove dead ones
        for func in toremove:
            self._handlers.remove(func)
    def emit_now(self, *args, **kwargs):
        """emit_now(*args, **kwargs)
        Emit the signal *now*. All handlers are called from the calling
        thread. Beware, this should only be done from the same thread
        that runs the event loop.
        """
        # Add an event for each callback
        toremove = []
        for func in self._handlers:
            if func.isdead():
                toremove.append(func)
            else:
                func.call(*args, **kwargs)
        # Remove dead ones
        for func in toremove:
            self._handlers.remove(func)
class TheTimerThread(threading.Thread):
    """TheTimerThread is a singleton thread that is used by all timers
    and delayed events to wait for a while (in a separate thread) and then
    post an event to the event-queue. By sharing a single thread timers
    stay lightweight and there is no time spend on initializing or tearing
    down threads. The downside is that when there are a lot of timers running
    at the same time, adding a timer may become a bit inefficient because
    the registered objects must be sorted each time an object is added.
    """
    def __init__(self):
        threading.Thread.__init__(self)
        self.setDaemon(True)
        self._exit = False
        self._timers = []
        self._somethingChanged = False
        self._condition = threading.Condition(threading.Lock())
    def stop(self, timeout=1.0):
        self._exit = True
        self._condition.acquire()
        try:
            self._condition.notify()
        finally:
            self._condition.release()
        self.join(timeout)
    def add(self, timer):
        """add(timer)
        Add item to the list of objects to track. The object should
        have a _timeout attribute, representing the time.time() at which
        it runs out, and an _on_timeout() method to call when it does.
        """
        # Check
        if not (hasattr(timer, "_timeout") and hasattr(timer, "_on_timeout")):
            raise ValueError("Cannot add this object to theTimerThread.")
        # Add item
        self._condition.acquire()
        try:
            if timer not in self._timers:
                self._timers.append(timer)
                self._sort()
                self._somethingChanged = True
            self._condition.notify()
        finally:
            self._condition.release()
    def _sort(self):
        self._timers = sorted(self._timers, key=lambda x: x._timeout,
reverse=True)
    def discard(self, timer):
        """Stop the timer if it hasn't finished yet"""
        self._condition.acquire()
        try:
            if timer in self._timers:
                self._timers.remove(timer)
            self._somethingChanged = True
            self._condition.notify()
        finally:
            self._condition.release()
    def run(self):
        self._condition.acquire()
        try:
            self._mainloop()
        finally:
            self._condition.release()
    def _mainloop(self):
        while not self._exit:
            # Set flag
            self._somethingChanged = False
            # Wait here, in wait() the undelying lock is released
            if self._timers:
                timer = self._timers[-1]
                timeout = timer._timeout - time.time()
                if timeout > 0:
                    self._condition.wait(timeout)
            else:
                timer = None
                self._condition.wait()
            # Here the lock has been re-acquired. Take action?
            if self._exit:
                break
            if (timer is not None) and (not self._somethingChanged):
                if timer._on_timeout():
                    self._sort()  # Keep and resort
                else:
                    self._timers.pop()  # Pop
# Instantiate and start the single timer thread
# We can do this as long as we do not wait for the threat, and the threat
# does not do any imports:
# http://docs.python.org/library/threading.html#importing-in-threaded-code
theTimerThread = TheTimerThread()
theTimerThread.start()
class Timer(Signal):
    """Timer(interval=1.0, oneshot=True)
    Timer class. You can bind callbacks to the timer. The timer is
    fired when it runs out of time.
    Parameters
    ----------
    interval : number
        The interval of the timer in seconds.
    oneshot : bool
        Whether the timer should do a single shot, or run continuously.
    """
    def __init__(self, interval=1.0, oneshot=True):
        Signal.__init__(self)
        # store Timer specific properties
        self.interval = interval
        self.oneshot = oneshot
        #
        self._timeout = 0
    @Property
    def interval():
        """Set/get the timer's interval in seconds."""
        def fget(self):
            return self._interval
        def fset(self, value):
            if not isinstance(value, (int, float)):
                raise ValueError("interval must be a float or integer.")
            if value <= 0:
                raise ValueError("interval must be larger than 0.")
            self._interval = float(value)
        return locals()
    @Property
    def oneshot():
        """Set/get whether this is a oneshot timer. If not is runs
        continuously.
        """
        def fget(self):
            return self._oneshot
        def fset(self, value):
            self._oneshot = bool(value)
        return locals()
    @property
    def running(self):
        """Get whether the timer is running."""
        return self._timeout > 0
    def start(self, interval=None, oneshot=None):
        """start(interval=None, oneshot=None)
        Start the timer. If interval or oneshot are not given,
        their current values are used.
        """
        # set properties?
        if interval is not None:
            self.interval = interval
        if oneshot is not None:
            self.oneshot = oneshot
        # put on
        self._timeout = time.time() + self.interval
        theTimerThread.add(self)
    def stop(self):
        """stop()
        Stop the timer from running.
        """
        theTimerThread.discard(self)
        self._timeout = 0
    def _on_timeout(self):
        """Method to call when the timer finishes. Called from
        event-loop-thread.
        """
        # Emit signal
        self.emit()
        # print('timer timeout', self.oneshot)
        # Do we need to stop it now, or restart it
        if self.oneshot:
            # This timer instance is removed from the list of Timers
            # when the timeout is reached.
            self._timeout = 0
            return False
        else:
            # keep in the thread
            self._timeout = time.time() + self.interval
            return True
class YotonApplication(object):
    """YotonApplication
    Represents the yoton application and contains functions for
    the event system. Multiple instances can be created, they will
    all operate on the same event queue and share attributes
    (because these are on the class, not on the instance).
    One instance of this class is always accesible via yoton.app.
    For convenience, several of its methods are also accessible
    directly from the yoton module namespace.
    """
    # Event queues
    _event_queue = PackageQueue(10000, "new")
    # Flag to stop event loop
    _stop_event_loop = False
    # Flag to signal whether we are in an event loop
    # Can be set externally if the event loop is hijacked.
    _in_event_loop = False
    # To allow other event loops to embed the yoton event loop
    _embedding_callback1 = None  # The reference
    _embedding_callback2 = None  # Used in post_event
    def call_later(self, func, timeout=0.0, *args, **kwargs):
        """call_later(func, timeout=0.0, *args, **kwargs)
        Call the given function after the specified timeout.
        Parameters
        ----------
        func : callable
            The function to call.
        timeout : number
            The time to wait in seconds. If zero, the event is put on the event
            queue. If negative, the event will be put at the front of the event
            queue, so that it's processed asap.
        args : arguments
            The arguments to call func with.
        kwargs: keyword arguments.
```

```
            The keyword arguments to call func with.
        """
        # Wrap the object in an event
        event = Event(func, *args, **kwargs)
        # Put it in the queue
        if timeout > 0:
            self.post_event_later(event, timeout)
        elif timeout < 0:
            self.post_event_asap(event)  # priority event
        else:
            self.post_event(event)
    def post_event(self, event):
        """post_event(events)
        Post an event to the event queue.
        """
        YotonApplication._event_queue.push(event)
        #
        if YotonApplication._embedding_callback2 is not None:
            YotonApplication._embedding_callback2 = None
            YotonApplication._embedding_callback1()
    def post_event_asap(self, event):
        """post_event_asap(event)
        Post an event to the event queue. Handle as soon as possible;
        putting it in front of the queue.
        """
        YotonApplication._event_queue.insert(event)
        #
        if YotonApplication._embedding_callback2 is not None:
            YotonApplication._embedding_callback2 = None
            YotonApplication._embedding_callback1()
    def post_event_later(self, event, delay):
        """post_event_later(event, delay)
        Post an event to the event queue, but with a certain delay.
        """
        event._timeout = time.time() + delay
        theTimerThread.add(event)
        # Calls post_event in due time
    def process_events(self, block=False):
        """process_events(block=False)
        Process all yoton events currently in the queue.
        This function should be called periodically
        in order to keep the yoton event system running.
        block can be False (no blocking), True (block), or a float
        blocking for maximally 'block' seconds.
        """
        # Reset callback for the embedding event loop
        YotonApplication._embedding_callback2 = 
YotonApplication._embedding_callback1
        # Process events
        try:
            while True:
                event = YotonApplication._event_queue.pop(block)
                event.dispatch()
                block = False  # Proceed until there are now more events
        except PackageQueue.Empty:
            pass
    def start_event_loop(self):
        """start_event_loop()
        Enter an event loop that keeps calling yoton.process_events().
        The event loop can be stopped using stop_event_loop().
        """
        # Dont go if we are in an event loop
        if YotonApplication._in_event_loop:
            return
        # Set flags
        YotonApplication._stop_event_loop = False
        YotonApplication._in_event_loop = True
        try:
            # Keep blocking for 3 seconds so a keyboardinterrupt still works
            while not YotonApplication._stop_event_loop:
                self.process_events(3.0)
        finally:
            # Unset flag
            YotonApplication._in_event_loop = False
    def stop_event_loop(self):
        """stop_event_loop()
        Stops the event loop if it is running.
        """
        if not YotonApplication._stop_event_loop:
            # Signal stop
            YotonApplication._stop_event_loop = True
            # Push an event so that process_events() unblocks
            def dummy():
                pass
            self.post_event(Event(dummy))
    def embed_event_loop(self, callback):
        """embed_event_loop(callback)
        Embed the yoton event loop in another event loop. The given callback
        is called whenever a new yoton event is created. The callback
        should create an event in the other event-loop, which should
        lead to a call to the process_events() method. The given callback
        should be thread safe.
        Use None as an argument to disable the embedding.
        """
        YotonApplication._embedding_callback1 = callback
        YotonApplication._embedding_callback2 = callback
# Instantiate an application object
app = YotonApplication()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module yoton.misc
Defines a few basic constants, classes and functions.
Most importantly, it defines a couple of specific buffer classes that
are used for the low-level messaging.
"""
import sys, time
import struct
import socket
import threading
import random
from collections import deque
# Version dependent defs
V2 = sys.version_info[0] == 2
if V2:
    if sys.platform.startswith("java"):
        import __builtin__ as D  # Jython
    else:
        D = __builtins__
    if not isinstance(D, dict):
        D = D.__dict__
    bytes = D["str"]
    str = D["unicode"]
    xrange = D["xrange"]
    basestring = basestring  # noqa
    long = long  # noqa
else:
    basestring = str  # to check if instance is string
    bytes, str = bytes, str
    long = int  # for the port
    xrange = range
def Property(function):
    """Property(function)
    A property decorator which allows to define fget, fset and fdel
    inside the function.
    Note that the class to which this is applied must inherit from object!
    Code based on an example posted by Walker Hale:
    http://code.activestate.com/recipes/410698/#c6
    """
    # Define known keys
    known_keys = "fget", "fset", "fdel", "doc"
    # Get elements for defining the property. This should return a dict
    func_locals = function()
    if not isinstance(func_locals, dict):
        raise RuntimeError('Property function should "return locals()".')
    # Create dict with kwargs for property(). Init doc with docstring.
    D = {"doc": function.__doc__}
    # Copy known keys. Check if there are invalid keys
    for key in func_locals.keys():
        if key in known_keys:
            D[key] = func_locals[key]
        else:
            raise RuntimeError("Invalid Property element: %s" % key)
    # Done
    return property(**D)
def getErrorMsg():
    """getErrorMsg()
    Return a string containing the error message. This is usefull, because
    there is no uniform way to catch exception objects in Python 2.x and
    Python 3.x.
    """
    # Get traceback info
    type, value, tb = sys.exc_info()
    # Store for debugging?
    if True:
        sys.last_type = type
        sys.last_value = value
        sys.last_traceback = tb
    # Print
    err = ""
    try:
        if not isinstance(value, (OverflowError, SyntaxError, ValueError)):
            while tb:
                err = "line %i of %s." % (
                    tb.tb_frame.f_lineno,
                    tb.tb_frame.f_code.co_filename,
                )
                tb = tb.tb_next
    finally:
        del tb
    return str(value) + "\n" + err
def slot_hash(name):
    """slot_hash(name)
    Given a string (the slot name) returns a number between 8 and 2**64-1
    (just small enough to fit in a 64 bit unsigned integer). The number
    is used as a slot id.
    Slots 0-7 are reseved slots.
    """
    fac = 0xD2D84A61
    val = 0
    offset = 8
    for c in name:
        val += (val >> 3) + (ord(c) * fac)
    val += (val >> 3) + (len(name) * fac)
    return offset + (val % (2**64 - offset))
def port_hash(name):
    """port_hash(name)
    Given a string, returns a port number between 49152 and 65535.
    (2**14 (16384) different posibilities)
    This range is the range for dynamic and/or private ports
    (ephemeral ports) specified by iana.org.
    The algorithm is deterministic, thus providing a way to map names
    to port numbers.
    """
    fac = 0xD2D84A61
    val = 0
    for c in name:
        val += (val >> 3) + (ord(c) * fac)
    val += (val >> 3) + (len(name) * fac)
    return 49152 + (val % 2**14)
def split_address(address):
    """split_address(address) -> (protocol, hostname, port)
    Split address in protocol, hostname and port. The address has the
    following format: "protocol://hostname:port". If the protocol is
    omitted, TCP is assumed.
    The hostname is the name or ip-address of the computer to connect to.
    One can use "localhost" for a connection that bypasses some
    network layers (and is not visible from the outside). One can use
    "publichost" for a connection at the current computers IP address
    that is visible from the outside.
    The port can be an integer, or a sting. In the latter case the integer
    port number is calculated using a hash. One can also use "portname+offset"
    to specify an integer offset for the port number.
    """
    # Check
    if not isinstance(address, basestring):
        raise ValueError("Address should be a string.")
    if ":" not in address:
        raise ValueError("Address should be in format 'host:port'.")
    # Is the protocol explicitly defined (zeromq compatibility)
    protocol = ""
    if "://" in address:
        # Get protocol and stripped address
        tmp = address.split("://", 1)
        protocol = tmp[0].lower()
        address = tmp[1]
    if not protocol:
        protocol = "tcp"
    # Split
    tmp = address.split(":", 1)
    host, port = tuple(tmp)
    # Process host
    if host.lower() == "localhost":
        host = "127.0.0.1"
    if host.lower() == "publichost":
        host = "publichost" + "0"
    if host.lower().startswith("publichost") and host[10:] in "0123456789":
        index = int(host[10:])
        hostname = socket.gethostname()
        tmp = socket.gethostbyname_ex(hostname)
        try:
            host = tmp[2][index]  # This resolves to 127.0.1.1 on some Linuxes
        except IndexError:
            raise ValueError("Invalid index (%i) in public host addresses." %
index)
    # Process port
    try:
        port = int(port)
    except ValueError:
        # Convert to int, using a hash
        # Is there an offset?
        offset = 0
        if "+" in port:
            tmp = port.split("+", 1)
            port, offset = tuple(tmp)
            try:
                offset = int(offset)
            except ValueError:
                raise ValueError("Invalid offset in address")
        # Convert
        port = port_hash(port) + offset
    # Check port
    # if port < 1024 or port > 2**16:
    #    raise ValueError("The port must be in the range [1024, 2^16>.")
    if port > 2**16:
        raise ValueError("The port must be in the range [0, 2^16>.")
    # Done
    return protocol, host, port
class UID:
    """UID
    Represents an 8-byte (64 bit) Unique Identifier.
    """
    _last_timestamp = 0
    def __init__(self, id=None):
        # Create nr consisting of two parts
        if id is None:
            self._nr = self._get_time_int() << 32
            self._nr += self._get_random_int()
        elif isinstance(id, (int, long)):
            self._nr = id
        else:
            raise ValueError("The id given to UID() should be an int.")
    def __repr__(self):
        h = self.get_hex()
        return "<UID %s-%s>" % (h[:8], h[8:])
    def get_hex(self):
        """get_hex()
        Get the hexadecimal representation of this UID. The returned
        string is 16 characters long.
        """
        h = hex(self._nr)
        h = h[2:].rstrip("L")
        h = h.ljust(2 * 8, "0")
        return h
    def get_bytes(self):
        """get_bytes()
        Get the UID as bytes.
        """
        return struct.pack("<Q", self._nr)
    def get_int(self):
        """get_int()
        Get the UID as a 64 bit (long) integer.
        """
        return self._nr
    def _get_random_int(self):
        return random.randrange(0xFFFFFFFF)
    def _get_time_int(self):
        # Get time stamp in steps of miliseconds
        timestamp = int(time.time() * 1000)
        # Increase by one if the same as last time
        if timestamp <= UID._last_timestamp:
            timestamp = UID._last_timestamp + 1
        # Store for next time
        UID._last_timestamp = timestamp
        # Truncate to 4 bytes. If the time goes beyond the integer limit, we
just
        # restart counting. With this setup, the cycle is almost 25 days
        timestamp = timestamp & 0xFFFFFFFF
        # Don't allow 0
        if timestamp == 0:
            timestamp += 1
            UID._last_timestamp += 1
        return timestamp
class PackageQueue(object):
    """PackageQueue(N, discard_mode='old')
    A queue implementation that can be used in blocking and non-blocking
    mode and allows peeking. The queue has a limited size. The user
    can specify whether old or new messages should be discarted.
    Uses a deque object for the queue and a threading.Condition for
    the blocking.
    """
    class Empty(Exception):
        def __init__(self):
            Exception.__init__(self, "pop from an empty PackageQueue")
        pass
    def __init__(self, N, discard_mode="old"):
        # Instantiate queue and condition
        self._q = deque()
        self._condition = threading.Condition()
        # Store max number of elements in queue
        self._maxlen = int(N)
        # Store discard mode as integer
        discard_mode = discard_mode.lower()
        if discard_mode == "old":
            self._discard_mode = 1
        elif discard_mode == "new":
            self._discard_mode = 2
        else:
            raise ValueError("Invalid discard mode.")
    def full(self):
        """full()
        Returns True if the number of elements is at its maximum right now.
        Note that in theory, another thread might pop an element right
        after this function returns.
        """
        return len(self) >= self._maxlen
    def empty(self):
        """empty()
        Returns True if the number of elements is zero right now. Note
        that in theory, another thread might add an element right
        after this function returns.
        """
        return len(self) == 0
    def push(self, x):
        """push(item)
        Add an item to the queue. If the queue is full, the oldest
        item in the queue, or the given item is discarted.
        """
        condition = self._condition
        condition.acquire()
        try:
            q = self._q
            if len(q) < self._maxlen:
                # Add now and notify any waiting threads in get()
                q.append(x)
                condition.notify()  # code at wait() procedes
            else:
                # Full, either discard or pop (no need to notify)
                if self._discard_mode == 1:
                    q.popleft()  # pop old
                    q.append(x)
                elif self._discard_mode == 2:
                    pass  # Simply do not add
        finally:
            condition.release()
    def insert(self, x):
        """insert(x)
        Insert an item at the front of the queue. A call to pop() will
        get this item first. This should be used in rare circumstances
        to give an item priority. This method never causes items to
        be discarted.
        """
        condition = self._condition
        condition.acquire()
        try:
            self._q.appendleft(x)
            condition.notify()  # code at wait() procedes
        finally:
            condition.release()
    def pop(self, block=True):
        """pop(block=True)
        Pop the oldest item from the queue. If there are no items in the
        queue:
          * the calling thread is blocked until an item is available
            (if block=True, default);
          * an PackageQueue.Empty exception is raised (if block=False);
          * the calling thread is blocked for 'block' seconds (if block
            is a float).
        """
        condition = self._condition
        condition.acquire()
        try:
            q = self._q
            if not block:
                # Raise empty if no items in the queue
                if not len(q):
                    raise self.Empty()
            elif block is True:
                # Wait for notify (awakened does not guarantee len(q)>0)
                while not len(q):
                    condition.wait()
            elif isinstance(block, float):
                # Wait if no items, then raise error if still no items
                if not len(q):
                    condition.wait(block)
                    if not len(q):
                        raise self.Empty()
            else:
                raise ValueError("Invalid value for block in
PackageQueue.pop().")
```

```
            # Return item
            return q.popleft()
        finally:
            condition.release()
    def peek(self, index=0):
        """peek(index=0)
        Get an item from the queue without popping it. index=0 gets the
        oldest item, index=-1 gets the newest item. Note that index access
        slows to O(n) time in the middle of the queue (due to the undelying
        deque object).
        Raises an IndexError if the index is out of range.
        """
        return self._q[index]
    def __len__(self):
        return self._q.__len__()
    def clear(self):
        """clear()
        Remove all items from the queue.
        """
        self._condition.acquire()
        try:
            self._q.clear()
        finally:
            self._condition.release()
class TinyPackageQueue(PackageQueue):
    """TinyPackageQueue(N1, N2, discard_mode='old', timeout=1.0)
    A queue implementation that can be used in blocking and non-blocking
    mode and allows peeking. The queue has a tiny-size (N1). When this size
    is reached, a call to push() blocks for up to timeout seconds. The
    real size (N2) is the same as in the PackageQueue class.
    The tinysize mechanism can be used to semi-synchronize a consumer
    and a producer, while still having a small queue and without having
    the consumer fully block.
    Uses a deque object for the queue and a threading.Condition for
    the blocking.
    """
    def __init__(self, N1, N2, discard_mode="old", timeout=1.0):
        PackageQueue.__init__(self, N2, discard_mode)
        # Store limit above which the push() method will block
        self._tinylen = int(N1)
        # Store timeout
        self._timeout = timeout
    def push(self, x):
        """push(item)
        Add an item to the queue. If the queue has >= n1 values,
        this function will block timeout seconds, or until an item is
        popped from another thread.
        """
        condition = self._condition
        condition.acquire()
        try:
            q = self._q
            lq = len(q)
            if lq < self._tinylen:
                # We are on safe side. Wake up any waiting threads if queue was
empty
                q.append(x)
                condition.notify()  # pop() at wait() procedes
            elif lq < self._maxlen:
                # The queue is above its limit, but not full
                condition.wait(self._timeout)
                q.append(x)
            else:
                # Full, either discard or pop (no need to notify)
                if self._discard_mode == 1:
                    q.popleft()  # pop old
                    q.append(x)
                elif self._discard_mode == 2:
                    pass  # Simply do not add
        finally:
            condition.release()
    def pop(self, block=True):
        """pop(block=True)
        Pop the oldest item from the queue. If there are no items in the
        queue:
          * the calling thread is blocked until an item is available
            (if block=True, default);
          * a PackageQueue.Empty exception is raised (if block=False);
          * the calling thread is blocked for 'block' seconds (if block
            is a float).
        """
        condition = self._condition
        condition.acquire()
        try:
            q = self._q
            if not block:
                # Raise empty if no items in the queue
                if not len(q):
                    raise self.Empty()
            elif block is True:
                # Wait for notify (awakened does not guarantee len(q)>0)
                while not len(q):
                    condition.wait()
            elif isinstance(block, float):
                # Wait if no items, then raise error if still no items
                if not len(q):
                    condition.wait(block)
                    if not len(q):
                        raise self.Empty()
            else:
                raise ValueError("Invalid value for block in 
PackageQueue.pop().")
            # Notify if this pop would reduce the length below the threshold
            if len(q) <= self._tinylen:
                if sys.version_info < (2, 6):
                    condition.notifyAll()  # wait() procedes
                else:
                    condition.notify_all()
            # Return item
            return q.popleft()
        finally:
            condition.release()
    def clear(self):
        """clear()
        Remove all items from the queue.
        """
        self._condition.acquire()
        try:
            lq = len(self._q)
            self._q.clear()
            if lq >= self._tinylen:
                self._condition.notify()
        finally:
            self._condition.release()
```

```
# -*- coding: utf-8 -*-
# flake8: noqa
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
"""
Yoton is a Python package that provides a simple interface
to communicate between two or more processes.
Yoton is ...
  * lightweight
  * written in pure Python
  * without dependencies (except Python)
  * available on Python version >= 2.4, including Python 3
  * cross-platform
  * pretty fast
"""
# Import stuff from misc and events
from yoton.misc import UID, str, bytes
from yoton.events import Signal, Timer, app
# Inject app function in yoton namespace for convenience
call_later = app.call_later
process_events = app.process_events
start_event_loop = app.start_event_loop
stop_event_loop = app.stop_event_loop
embed_event_loop = app.embed_event_loop
# Import more
from yoton.core import Package
from yoton.connection import Connection, ConnectionCollection
from yoton.connection_tcp import TcpConnection
from yoton.context import Context
from yoton.clientserver import RequestServer, do_request
from yoton.channels import *
# Set yoton version
__version__ = "2.2"
# Define convenience class
class SimpleSocket(Context):
    """SimpleSocket()
    A simple socket has an API similar to a BSD socket. This socket
    sends whole text messages from one end to the other.
    This class subclasses the Yoton.Context class, which makes setting
    up this socket very easy.
    Example
    -------
    # One end
    s = SimpleSocket()
    s.bind('localhost:test')
    s.send("Hi")
    # Other end
    s = SimpleSocket()
    s.connect('localhost:test')
    print(s.recv())
    """
    def __init__(self, verbose=False):
        Context.__init__(self, verbose)
        # Create channels
        self._cs = PubChannel(self, "text")
        self._cr = SubChannel(self, "text")
    def send(self, s):
        """send(message)
        Send a text message. The message is queued and send
        over the socket by the IO-thread.
        """
        self._cs.send(s)
    def recv(self, block=None):
        """recv(block=None):
        Read a text from the channel. What was send as one message is
        always received as one message.
        If the channel is closed and all messages are read, returns ''.
        """
        return self._cr.recv(block)
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module yoton.channels.channels_base
Defines the base channel class and the MessageType class.
"""
import sys
import time
import threading
import yoton
from yoton.misc import basestring
from yoton.misc import slot_hash, PackageQueue
from yoton.core import Package
from yoton.context import Context
from yoton.channels.message_types import MessageType, TEXT
class BaseChannel(object):
    """BaseChannel(context, slot_base, message_type=yoton.TEXT)
    Abstract class for all channels.
    Parameters
    ----------
    context : yoton.Context instance
        The context that this channel uses to send messages in a network.
    slot_base : string
        The base slot name. The channel appends an extension to indicate
        message type and messaging pattern to create the final slot name.
        The final slot is used to connect channels at different contexts
        in a network
    message_type : yoton.MessageType instance
        (default is yoton.TEXT)
        Object to convert messages to bytes and bytes to messages.
        Users can create their own message_type class to enable
        communicating any type of message they want.
    Details
    -------
    Messages send via a channel are delivered asynchronically to the
    corresponding channels.
    All channels are associated with a context and can be used to send
    messages to other channels in the network. Each channel is also
    associated with a slot, which is a string that represents a kind
    of address. A message send by a channel at slot X can only be received
    by a channel with slot X.
    Note that the channel appends an extension
    to the user-supplied slot name, that represents the message type
    and messaging pattern of the channel. In this way, it is prevented
    that for example a PubChannel can communicate with a RepChannel.
    """
    def __init__(self, context, slot_base, message_type=None):
        # Store context
        if not isinstance(context, Context):
            raise ValueError("Context not valid.")
        self._context = context
        # Check message type
        if message_type is None:
            message_type = TEXT
        if isinstance(message_type, type) and issubclass(message_type,
MessageType):
            message_type = message_type()
        if isinstance(message_type, MessageType):
            message_type = message_type
        else:
            raise ValueError("message_type should be a MessageType instance.")
        # Store message type and conversion methods
        self._message_type_instance = message_type
        self.message_from_bytes = message_type.message_from_bytes
        self.message_to_bytes = message_type.message_to_bytes
        # Queue for incoming trafic (not used for pure sending channels)
        self._q_in = PackageQueue(*context._queue_params)
        # For sending channels: to lock the channel for sending
        self._send_condition = threading.Condition()
        self._is_send_locked = 0  # "True" is the timeout time
        # Signal for receiving data
        self._received_signal = yoton.events.Signal()
        self._posted_received_event = False
        # Channels can be closed
        self._closed = False
        # Event driven mode
        self._run_mode = 0
        # Init slots
        self._init_slots(slot_base)
    def _init_slots(self, slot_base):
        """_init_slots(slot_base)
        Called from __init__ to initialize the slots and perform all checks.
        """
        # Check if slot is string
        if not isinstance(slot_base, basestring):
            raise ValueError("slot_base must be a string.")
        # Get full slot names, init byte versions
        slots_t = []
        slots_h = []
        # Get extension for message type and messaging pattern
        ext_type = self._message_type_instance.message_type_name()
        ext_patterns = self._messaging_patterns()  # (incoming, outgoing)
        # Normalize and check slot names
        for ext_pattern in ext_patterns:
            if not ext_pattern:
                slots_t.append(None)
                slots_h.append(0)
                continue
            # Get full name
            slot = slot_base + "." + ext_type + "." + ext_pattern
            # Store text version
            slots_t.append(slot)
            # Strip and make lowercase
            slot = slot.strip().lower()
            # Hash
            slots_h.append(slot_hash(slot))
        # Store slots
        self._slot_out = slots_t[0]
        self._slot_in = slots_t[1]
        self._slot_out_h = slots_h[0]
        self._slot_in_h = slots_h[1]
        # Register slots (warn if neither slot is valid)
        if self._slot_out_h:
            self._context._register_sending_channel(
                self, self._slot_out_h, self._slot_out
            )
        if self._slot_in_h:
            self._context._register_receiving_channel(
                self, self._slot_in_h, self._slot_in
            )
        if not self._slot_out_h and not self._slot_in_h:
            raise ValueError("This channel does not have valid slots.")
    def _messaging_patterns(self):
        """_messaging_patterns()
        Implement to return a string that specifies the pattern
        for sending and receiving, respecitively.
        """
        raise NotImplementedError()
    def close(self):
        """close()
        Close the channel, i.e. unregisters this channel at the context.
        A closed channel cannot be reused.
        Future attempt to send() messages will result in an IOError
        being raised. Messages currently in the channel's queue can
        still be recv()'ed, but no new messages will be delivered at
        this channel.
        """
        # We keep a reference to the context, otherwise we need locks
        # The context clears the reference to this channel when unregistering.
        self._closed = True
        self._context._unregister_channel(self)
    def _send(self, message, dest_id=0, dest_seq=0):
        """_send(message, dest_id=0, dest_seq=0)
        Sends a message of raw bytes without checking whether they're bytes.
        Optionally, dest_id and dest_seq represent the message that
        this message  replies to. These are used for the request/reply
        pattern.
        Returns the package that will be send (or None). The context
        will set _source_id on the package right before
        sending it away.
        """
        # Check if still open
        if self._closed:
            className = self.__class__.__name__
            raise IOError("Cannot send from closed %s %i." % (className,
id(self)))
        if message:
            # If send_locked, wait at most one second
            if self._is_send_locked:
                self._send_condition.acquire()
                try:
                    self._send_condition.wait(1.0)  # wait for notify
                finally:
                    self._send_condition.release()
                    if time.time() > self._is_send_locked:
                        self._is_send_locked = 0
            # Push it on the queue as a package
            slot = self._slot_out_h
            cid = self._context._id
            p = Package(message, slot, cid, 0, dest_id, dest_seq, 0)
            self._context._send_package(p)
            # Return package
            return p
        else:
            return None
    def _recv(self, block):
        """_recv(block)
        Receive a package (or None).
        """
        if block is True:
            # Block for 0.25 seconds so that KeyboardInterrupt works
            while not self._closed:
                try:
                    return self._q_in.pop(0.25)
                except self._q_in.Empty:
                    continue
        else:
            # Block normal
            try:
                return self._q_in.pop(block)
            except self._q_in.Empty:
                return None
    def _set_send_lock(self, value):
        """_set_send_lock(self, value)
        Set or unset the blocking for the _send() method.
        """
        # Set send lock variable. We adopt a timeout (10s) just in case
        # the SubChannel that locks the PubChannel gets disconnected and
        # is unable to unlock it.
        if value:
            self._is_send_locked = time.time() + 10.0
        else:
            self._is_send_locked = 0
        # Notify any threads that are waiting in _send()
        if not value:
            self._send_condition.acquire()
            try:
                if sys.version_info < (2, 6):
                    self._send_condition.notifyAll()
                else:
                    self._send_condition.notify_all()
            finally:
                self._send_condition.release()
    ## How packages are inserted in this channel for receiving
    def _inject_package(self, package):
        """_inject_package(package)
        Same as _recv_package, but by definition do not block.
        _recv_package is overloaded in SubChannel. _inject_package is not.
        """
        self._q_in.push(package)
        self._maybe_emit_received()
    def _recv_package(self, package):
        """_recv_package(package)
        Put package in the queue.
        """
        self._q_in.push(package)
        self._maybe_emit_received()
    def _maybe_emit_received(self):
        """_maybe_emit_received()
        We want to emit a signal, but in such a way that multiple
        arriving packages result in a single emit. This methods
        only posts an event if it has not been done, or if the previous
        event has been handled.
        """
        if not self._posted_received_event:
            self._posted_received_event = True
            event = yoton.events.Event(self._emit_received)
            yoton.app.post_event(event)
    def _emit_received(self):
        """_emit_received()
        Emits the "received" signal. This method is called once new data
        has been received. However, multiple arrived messages may
        result in a single call to this method. There is also no
        guarantee that recv() has not been called in the mean time.
        Also sets the variabele so that a new event for this may be
        created. This method is called from the event loop.
        """
        self._posted_received_event = False  # Reset
        self.received.emit_now(self)
    # Received property sits on the BaseChannel because is is used by almost
    # all channels. Note that PubChannels never emit this signal as they
    # catch status messages from the SubChannel by overloading _recv_package().
    @property
    def received(self):
        """Signal that is emitted when new data is received. Multiple
        arrived messages may result in a single call to this method.
        There is no guarantee that recv() has not been called in the
        mean time. The signal is emitted with the channel instance
        as argument.
        """
        return self._received_signal
    ## Properties
    @property
    def pending(self):
        """Get the number of pending incoming messages."""
        return len(self._q_in)
    @property
    def closed(self):
        """Get whether the channel is closed."""
        return self._closed
    @property
    def slot_outgoing(self):
        """Get the outgoing slot name."""
        return self._slot_out
    @property
    def slot_incoming(self):
        """Get the incoming slot name."""
        return self._slot_in
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module yoton.channels.file
Defines a class that can be used to wrap a channel to give it
a file like interface.
"""
import sys
import os
from yoton.channels import PubChannel, SubChannel
PY2 = sys.version_info[0] == 2
class FileWrapper(object):
    """FileWrapper(channel, chunksize=0, echo=None)
    Class that wraps a PubChannel or SubChannel instance to provide
    a file-like interface by implementing methods such as read() and
    write(), and other stuff specified in:
    [[http://docs.python.org/library/stdtypes.html#bltin-file-objects]]
    The file wrapper also splits messages into smaller messages if they
    are above the chunksize (only if chunksize > 0).
    On Python 2, the read methods return str (utf-8 encoded Unicode).
    """
    # Our file-like objects should not implement:
    # explicitly stated: fileno, isatty
    # don't seem to make sense: readlines, seek, tell, truncate, errors,
    # mode, name,
    def __init__(self, channel, chunksize=0, echo=None, isatty=False):
        if not isinstance(channel, (PubChannel, SubChannel)):
            raise ValueError("FileWrapper needs a PubChannel or SubChannel.")
        if echo is not None:
            if not isinstance(echo, PubChannel):
                raise ValueError("FileWrapper echo needs to be a PubChannel.")
        self._channel = channel
        self._chunksize = int(chunksize)
        self._echo = echo
        self._pid = os.getpid()  # To detect whether we are in multi-process
        self.errors = "strict"  # compat
        self._isatty = isatty
    def close(self):
        """Close the file object."""
        # Deal with multiprocessing
        if self._pid != os.getpid():
            if self is sys.stdin:
                sys.__stdin__.close()
            elif self is sys.stdout:
                sys.__stdout__.close()
            elif self is sys.stderr:
                sys.__stderr__.close()
            return
        # Normal behavior
        self._channel.close()
    @property
    def encoding(self):
        """The encoding used to encode strings to bytes and vice versa."""
        return "UTF-8"
    @property
    def closed(self):
        """Get whether the file is closed."""
        return self._channel._closed
    def flush(self):
        """flush()
        Wait here until all messages have been send.
        """
        self._channel._context.flush()
    @property
    def newlines(self):
        """The type of newlines used. Returns None; we never know what the
        other end could be sending!
        """
        return None
    # this is for the print statement to keep track spacing stuff
    def _set_softspace(self, value):
        self._softspace = bool(value)
    def _get_softspace(self):
        return hasattr(self, "_softspace") and self._softspace
    softspace = property(_get_softspace, _set_softspace, None, "")
    def read(self, block=None):
        """read(block=None)
        Alias for recv().
        """
        res = self._channel.recv(block)
        if res and self._echo is not None:
            self._echo.send(res)
        if PY2:
            return res.encode("utf-8")
        else:
            return res
    def write(self, message):
        """write(message)
        Uses channel.send() to send the message over the Yoton network.
        The message is partitioned in smaller parts if it is larger than
        the chunksize.
        """
        # Deal with multiprocessing
        if self._pid != os.getpid():
            realfile = None
            if self is sys.stdout:
                realfile = sys.__stdout__
            elif self is sys.stderr:
                realfile = sys.__stderr__
            if realfile is not None:
                sys.__stderr__.write(message)
                sys.__stderr__.flush()
            return
        chunkSize = self._chunksize
        if chunkSize > 0 and chunkSize < len(message):
            for i in range(0, len(message), chunkSize):
                self._channel.send(message[i : i + chunkSize])
        else:
            self._channel.send(message)
    def writelines(self, lines):
        """writelines(lines)
        Write a sequence of messages to the channel.
        """
        for line in lines:
            self._channel.send(line)
    def readline(self, size=0):
        """readline(size=0)
        Read one string that was send as one from the other end (always
        in blocking mode). A newline character is appended if it does not
        end with one.
        If size is given, returns only up to that many characters, the rest
        of the message is thrown away.
        """
        # Get line
        line = self._channel.recv(True)
        # Echo
        if line and self._echo is not None:
            self._echo.send(line)
        # Make sure it ends with newline
        if not line.endswith("\n"):
            line += "\n"
        # Decrease size?
        if size:
            line = line[:size]
        # Done
        if PY2:
            return line.encode("utf-8")
        else:
            return line
    def isatty(self):
        """Get whether this is a terminal."""
        return self._isatty
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module yoton.channels.channels_pubsub
Defines the channel classes for the pub/sub pattern.
"""
import time
from yoton.misc import bytes, xrange
from yoton.channels import BaseChannel
QUEUE_NULL = 0
QUEUE_OK = 1
QUEUE_FULL = 2
class PubChannel(BaseChannel):
    """PubChannel(context, slot_base, message_type=yoton.TEXT)
    The publish part of the publish/subscribe messaging pattern.
    Sent messages are received by all yoton.SubChannel instances with
    the same slot.
    There are no limitations for this channel if events are not processed.
    Parameters
    ----------
    context : yoton.Context instance
        The context that this channel uses to send messages in a network.
    slot_base : string
        The base slot name. The channel appends an extension to indicate
        message type and messaging pattern to create the final slot name.
        The final slot is used to connect channels at different contexts
        in a network
    message_type : yoton.MessageType instance
        (default is yoton.TEXT)
        Object to convert messages to bytes and bytes to messages.
        Users can create their own message_type class to let channels
        any type of message they want.
    """
    def __init__(self, *args, **kwargs):
        BaseChannel.__init__(self, *args, **kwargs)
        self._source_set = set()
    def _messaging_patterns(self):
        return "pub-sub", "sub-pub"
    def send(self, message):
        """send(message)
        Send a message over the channel. What is send as one
        message will also be received as one message.
        The message is queued and delivered to all corresponding
        SubChannels (i.e. with the same slot) in the network.
        """
        self._send(self.message_to_bytes(message))
    def _recv_package(self, package):
        """Overloaded to set blocking mode.
        Do not call _maybe_emit_received(), a PubChannel never emits
        the "received" signal.
        """
        message = package._data.decode("utf-8")
        source_id = package._source_id
        # Keep track of who's queues are full
        if message == "full":
            self._source_set.add(source_id)
        else:
            self._source_set.discard(source_id)
        # Set lock if there is a channel with a full queue,
        # Unset if there are none
        if self._source_set:
            self._set_send_lock(True)
            # sys.stderr.write('setting lock\n')
        else:
            self._set_send_lock(False)
            # sys.stderr.write('unsetting lock\n')
class SubChannel(BaseChannel):
    """SubChannel(context, slot_base, message_type=yoton.TEXT)
    The subscribe part of the publish/subscribe messaging pattern.
    Received messages were sent by a yoton.PubChannel instance at the
    same slot.
    This channel can be used as an iterator, which yields all pending
    messages. The function yoton.select_sub_channel can
    be used to synchronize multiple SubChannel instances.
    If no events being processed this channel works as normal, except
    that the received signal will not be emitted, and sync mode will
    not work.
    Parameters
    ----------
    context : yoton.Context instance
        The context that this channel uses to send messages in a network.
    slot_base : string
        The base slot name. The channel appends an extension to indicate
        message type and messaging pattern to create the final slot name.
        The final slot is used to connect channels at different contexts
        in a network
    message_type : yoton.MessageType instance
        (default is yoton.TEXT)
        Object to convert messages to bytes and bytes to messages.
        Users can create their own message_type class to let channels
        any type of message they want.
    """
    def __init__(self, *args, **kwargs):
        BaseChannel.__init__(self, *args, **kwargs)
        # To detect when to block the sending side
        self._queue_status = QUEUE_NULL
        self._queue_status_timeout = 0
        self._HWM = 32
        self._LWM = 16
        # Automatically check queue status when new data
        # enters the system
        self.received.bind(self._check_queue_status)
    def _messaging_patterns(self):
        return "sub-pub", "pub-sub"
    def __iter__(self):
        return self
    def __next__(self):  # Python 3.x
        m = self.recv(False)
        if m:
            return m
        else:
            raise StopIteration()
    def next(self):  # Python 2.x
        """next()
        Return the next message, or raises StopIteration if non available.
        """
        return self.__next__()
    ## For sync mode
    def set_sync_mode(self, value):
        """set_sync_mode(value)
        Set or unset the SubChannel in sync mode. When in sync mode, all
        channels that send messages to this channel are blocked if
        the queue for this SubChannel reaches a certain size.
        This feature can be used to limit the rate of senders if the consumer
        (i.e. the one that calls recv()) cannot keep up with processing
        the data.
        This feature requires the yoton event loop to run at the side
        of the SubChannel (not necessary for the yoton.PubChannel side).
        """
        value = bool(value)
        # First reset block status if necessary
        if self._queue_status == QUEUE_FULL:
            self._send_block_message_to_senders("ok")
        # Set new queue status flag
        if value:
            self._queue_status = QUEUE_OK
        else:
            self._queue_status = QUEUE_NULL
    def _send_block_message_to_senders(self, what):
        """_send_block_message_to_senders(what)
        Send a message to the PubChannel side to make it block/unblock.
        """
        # Check
        if not self._context.connection_count:
            return
        # Send
        try:
            self._send(what.encode("utf-8"))
        except IOError:
            # If self._closed
            self._check_queue_status = QUEUE_NULL
    def _check_queue_status(self, dummy=None):
        """_check_queue_status()
        Check the queue status. Returns immediately unless this receiving
        channel runs in sync mode.
        If the queue is above a certain size, will send out a package that
        will make the sending side block. If the queue is below a certain
        size, will send out a package that will make the sending side unblock.
        """
        if self._queue_status == QUEUE_NULL:
            return
        elif len(self._q_in) > self._HWM:
            if self._queue_status == QUEUE_OK:
                self._queue_status = QUEUE_FULL
                self._queue_status_timeout = time.time() + 4.0
                self._send_block_message_to_senders("full")
        elif len(self._q_in) < self._LWM:
            if self._queue_status == QUEUE_FULL:
                self._queue_status = QUEUE_OK
                self._queue_status_timeout = time.time() + 4.0
                self._send_block_message_to_senders("ok")
        # Resend every so often. After 10s the PubChannel will unlock itself
        if self._queue_status_timeout < time.time():
            self._queue_status_timeout = time.time() + 4.0
            if self._queue_status == QUEUE_OK:
                self._send_block_message_to_senders("ok")
            else:
                self._send_block_message_to_senders("full")
    ## Receive methods
    def recv(self, block=True):
        """recv(block=True)
        Receive a message from the channel. What was send as one
        message is also received as one message.
        If block is False, returns empty message if no data is available.
        If block is True, waits forever until data is available.
        If block is an int or float, waits that many seconds.
        If the channel is closed, returns empty message.
        """
        # Check queue status, maybe we need to block the sender
        self._check_queue_status()
        # Get package
        package = self._recv(block)
        # Return message content or None
        if package is not None:
            return self.message_from_bytes(package._data)
        else:
            return self.message_from_bytes(bytes())
    def recv_all(self):
        """recv_all()
        Receive a list of all pending messages. The list can be empty.
        """
        # Check queue status, maybe we need to block the sender
        self._check_queue_status()
        # Pop all messages and return as a list
        pop = self._q_in.pop
        packages = [pop() for i in xrange(len(self._q_in))]
        return [self.message_from_bytes(p._data) for p in packages]
    def recv_selected(self):
        """recv_selected()
        Receive a list of messages. Use only after calling
        yoton.select_sub_channel with this channel as one of the arguments.
        The returned messages are all received before the first pending
        message in the other SUB-channels given to select_sub_channel.
        The combination of this method and the function select_sub_channel
        enables users to combine multiple SUB-channels in a way that
        preserves the original order of the messages.
        """
        # No need to check queue status, we've done that in the
        # _get_pending_sequence_numbers() method
        # Prepare
        q = self._q_in
        ref_seq = self._ref_seq
        popped = []
        # Pop all messages that have sequence number lower than reference
        try:
            for i in xrange(len(q)):
                part = q.pop()
                if part._recv_seq > ref_seq:
                    q.insert(part)  # put back in queue
                    break
                else:
                    popped.append(part)
        except IndexError:
            pass
        # Done; return messages
        return [self.message_from_bytes(p._data) for p in popped]
    def _get_pending_sequence_numbers(self):
        """_get_pending_sequence_numbers()
        Get the sequence numbers of the first and last pending messages.
        Returns (-1,-1) if no messages are pending.
        Used by select_sub_channel() to determine which channel should
        be read from first and what the reference sequence number is.
        """
        # Check queue status, maybe we need to block the sender
        self._check_queue_status()
        # Peek
        try:
            q = self._q_in
            return q.peek(0)._recv_seq, q.peek(-1)._recv_seq + 1
        except IndexError:
            return -1, -1
def select_sub_channel(*args):
    """select_sub_channel(channel1, channel2, ...)
    Returns the channel that has the oldest pending message of all
    given yoton.SubCannel instances. Returns None if there are no pending
    messages.
    This function can be used to read from SubCannels instances in the
    order that the messages were send.
    After calling this function, use channel.recv_selected() to obtain
    all messages that are older than any pending messages in the other
    given channels.
    """
    # Init
    smallest_seq1 = 99999999999999999999999999
    smallest_seq2 = 99999999999999999999999999
    first_channel = None
    # For each channel ...
    for channel in args:
        # Check if channel is of right type
        if not isinstance(channel, SubChannel):
            raise ValueError("select_sub_channel() only accepts SUB channels.")
        # Get and check sequence
        seq1, seq2 = channel._get_pending_sequence_numbers()
        if seq1 >= 0:
            if seq1 < smallest_seq1:
                # Cannot go beyond number of packages in queue,
                # or than seq1 of earlier selected channel.
                smallest_seq2 = min(smallest_seq1, smallest_seq2, seq2)
                # Store
                smallest_seq1 = seq1
                first_channel = channel
            else:
                # The first_channel cannot go beyond the 1st package in THIS
queue
                smallest_seq2 = min(smallest_seq2, seq1)
    # Set flag at channel and return
    if first_channel:
        first_channel._ref_seq = smallest_seq2
        return first_channel
    else:
        return None
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module yoton.channels.channels_reqprep
Defines the channel classes for the req/rep pattern.
"""
import time
import threading
import yoton
from yoton.misc import basestring, bytes
from yoton.misc import getErrorMsg
from yoton.channels import BaseChannel, OBJECT
# For the req/rep channels to negotiate (simple load balancing)
REQREP_SEQ_REF = 2**63
# Define object to recognize errors
ERROR_OBJECT = "yoton_ERROR_HANDLING_REQUEST"
# Define exceoptions
class TimeoutError(Exception):
    pass
class CancelledError(Exception):
    pass
# # Try loading the exceptions from the concurrency framework
# # (or maybe not; it makes yoton less lightweight)
# try:
#     from concurrent.futures import TimeoutError, CancelledError
# except ImportError:
#     pass
class Future(object):
    """Future(req_channel, req, request_id)
    The Future object represents the future result of a request done at
    a yoton.ReqChannel.
    It enables:
      * checking whether the request is done.
      * getting the result or the exception raised during handling the request.
      * canceling the request (if it is not yet running)
      * registering callbacks to handle the result when it is available
    """
    def __init__(self, req_channel, req, request_id):
        # For being a Future object
        self._result = None
        self._status = 0  # 0:waiting, 1:running, 2:canceled, 3:error, 4:success
        self._callbacks = []
        # For handling req/rep
        self._req_channel = req_channel
        self._req = req
        self._request_id = request_id
        self._rep = bytes()
        self._replier = 0
        # For resending
        self._first_send_time = time.time()
        self._next_send_time = self._first_send_time + 0.5
        self._auto_cancel_timeout = 10.0
    def _send(self, msg):
        """_send(msg)
        For sending pre-request messages 'req?', 'req-'.
        """
        msg = msg.encode("utf-8")
        try:
            self._req_channel._send(msg, 0, self._request_id + REQREP_SEQ_REF)
        except IOError:
            # if self._closed, will call _send again, and catch IOerror,
            # which will result in one more call to cancel().
            self.cancel()
    def _resend_if_necessary(self):
        """_resend_if_necessary()
        Resends the pre-request message if we have not done so for the last
        0.5 second.
        This will also auto-cancel the message if it is resend over 20 times.
        """
        timetime = time.time()
        if self._status != 0:
            pass
        elif timetime > self._first_send_time + self._auto_cancel_timeout:
            self.cancel()
        elif timetime > self._next_send_time:
            self._send("req?")
            self._next_send_time = timetime + 0.5
    def set_auto_cancel_timeout(self, timeout):
        """set_auto_cancel_timeout(timeout):
        Set the timeout after which the call is automatically cancelled
        if it is not done yet. By default, this value is 10 seconds.
        If timeout is None, there is no limit to the wait time.
        """
        if timeout is None:
            timeout = 999999999999999999.0
        if timeout > 0:
            self._auto_cancel_timeout = float(timeout)
        else:
            raise ValueError("A timeout cannot be negative")
    def cancel(self):
        """cancel()
        Attempt to cancel the call. If the call is currently being executed
        and cannot be cancelled then the method will return False, otherwise
        the call will be cancelled and the method will return True.
        """
        if self._status == 1:
            # Running, cannot cancel
            return False
        elif self._status == 0:
            # Cancel now
            self._status = 2
            self._send("req-")
            for fn in self._callbacks:
                yoton.call_later(fn, 0, self)
            return True
        else:
            # Already done or canceled
            return True
    def cancelled(self):
        """cancelled()
        Return True if the call was successfully cancelled.
        """
        return self._status == 2
    def running(self):
        """running()
        Return True if the call is currently being executed and cannot be
        cancelled.
        """
        return self._status == 1
    def done(self):
        """done()
        Return True if the call was successfully cancelled or finished running.
        """
        return self._status in [2, 3, 4]
    def _wait(self, timeout):
        """_wait(timeout)
        Wait for the request to be handled for the specified amount of time.
        While waiting, the ReqChannel local event loop is called so that
        pre-request messages can be exchanged.
        """
        # No timout means a veeeery long timeout
        if timeout is None:
            timeout = 999999999999999999.0
        # Receive packages untill we receive the one we want,
        # or untill time runs out
        timestamp = time.time() + timeout
        while (self._status < 2) and (time.time() < timestamp):
            self._req_channel._process_events_local()
            time.sleep(0.01)  # 10 ms
    def result(self, timeout=None):
        """result(timeout=None)
        Return the value returned by the call. If the call hasn’t yet
        completed then this method will wait up to timeout seconds. If
        the call hasn’t completed in timeout seconds, then a TimeoutError
        will be raised. timeout can be an int or float. If timeout is not
        specified or None, there is no limit to the wait time.
        If the future is cancelled before completing then CancelledError
        will be raised.
        If the call raised, this method will raise the same exception.
        """
        # Wait
        self._wait(timeout)
        # Return or raise error
        if self._status < 2:
            raise TimeoutError("Result unavailable within the specified time.")
        elif self._status == 2:
            raise CancelledError("Result unavailable because request was 
cancelled.")
        elif self._status == 3:
            raise self._result
        else:
            return self._result
    def result_or_cancel(self, timeout=1.0):
        """result_or_cancel(timeout=1.0)
        Return the value returned by the call. If the call hasn’t yet
        completed then this method will wait up to timeout seconds. If
        the call hasn’t completed in timeout seconds, then the call is
        cancelled and the method will return None.
        """
        # Wait
        self._wait(timeout)
        # Return
        if self._status == 4:
            return self._result
        else:
            self.cancel()
            return None
    def exception(self, timeout=None):
        """exception(timeout)
        Return the exception raised by the call. If the call hasn’t yet
        completed then this method will wait up to timeout seconds. If
        the call hasn’t completed in timeout seconds, then a TimeoutError
        will be raised. timeout can be an int or float. If timeout is not
        specified or None, there is no limit to the wait time.
        If the future is cancelled before completing then CancelledError
        will be raised.
        If the call completed without raising, None is returned.
        """
        # Wait
        self._wait(timeout)
        # Return or raise error
        if self._status < 2:
            raise TimeoutError("Exception unavailable within the specified
time.")
        elif self._status == 2:
            raise CancelledError("Exception unavailable because request was
cancelled.")
        elif self._status == 3:
            return self._result
        else:
            return None  # no exception
    def add_done_callback(self, fn):
        """add_done_callback(fn)
        Attaches the callable fn to the future. fn will be called, with
        the future as its only argument, when the future is cancelled or
        finishes running.
        Added callables are called in the order that they were added. If
        the callable raises a Exception subclass, it will be logged and
        ignored. If the callable raises a BaseException subclass, the
        behavior is undefined.
        If the future has already completed or been cancelled, fn will be
        called immediately.
        """
        # Check
        if not hasattr(fn, "__call__"):
            raise ValueError("add_done_callback expects a callable.")
        # Add
        if self.done():
            yoton.call_later(fn, 0, self)
        else:
            self._callbacks.append(fn)
    def set_running_or_notify_cancel(self):
        """set_running_or_notify_cancel()
        This method should only be called by Executor implementations before
        executing the work associated with the Future and by unit tests.
        If the method returns False then the Future was cancelled, i.e.
        Future.cancel() was called and returned True.
        If the method returns True then the Future was not cancelled and
        has been put in the running state, i.e. calls to Future.running()
        will return True.
        This method can only be called once and cannot be called after
        Future.set_result() or Future.set_exception() have been called.
        """
        if self._status == 2:
            return False
        elif self._status == 0:
            self._status = 1
            return True
        else:
            raise RuntimeError(
                "set_running_or_notify_cancel should be called when in a clear
state."
            )
    def set_result(self, result):
        """set_result(result)
        Sets the result of the work associated with the Future to result.
        This method should only be used by Executor implementations and
        unit tests.
        """
        # Set result if indeed in running state
        if self._status == 1:
            self._result = result
            self._status = 4
            for fn in self._callbacks:
                yoton.call_later(fn, 0, self)
    def set_exception(self, exception):
        """set_exception(exception)
        Sets the result of the work associated with the Future to the
        Exception exception. This method should only be used by Executor
        implementations and unit tests.
        """
        # Check
        if isinstance(exception, basestring):
            exception = Exception(exception)
        if not isinstance(exception, Exception):
            raise ValueError("exception must be an Exception instance.")
        # Set result if indeed in running state
        if self._status == 1:
            self._result = exception
            self._status = 3
            for fn in self._callbacks:
                yoton.call_later(fn, 0, self)
class ReqChannel(BaseChannel):
    """ReqChannel(context, slot_base)
    The request part of the request/reply messaging pattern.
    A ReqChannel instance sends request and receive the corresponding
    replies. The requests are replied by a yoton.RepChannel instance.
    This class adopts req/rep in a remote procedure call (RPC) scheme.
    The handling of the result is done using a yoton.Future object, which
    follows the approach specified in PEP 3148. Note that for the use
    of callbacks, the yoton event loop must run.
    Basic load balancing is performed by first asking all potential
    repliers whether they can handle a request. The actual request
    is then send to the first replier to respond.
    Parameters
    ----------
    context : yoton.Context instance
        The context that this channel uses to send messages in a network.
    slot_base : string
        The base slot name. The channel appends an extension to indicate
        message type and messaging pattern to create the final slot name.
        The final slot is used to connect channels at different contexts
        in a network
    Usage
    -----
    One performs a call on a virtual method of this object. The actual
    method is executed by the yoton.RepChannel instance. The method can be
    called with normal and keyword arguments, which can be (a
    combination of): None, bool, int, float, string, list, tuple, dict.
    Example
    -------
    # Fast, but process is idling when waiting for the response.
    reply = req.add(3,4).result(2.0) # Wait two seconds
    # Asynchronous processing, but no waiting.
    def reply_handler(future):
        ... # Handle reply
    future = req.add(3,4)
    future.add_done_callback(reply_handler)
    """
    # Notes on load balancing:
    #
    # Firstly, each request has an id. Which is an integer number
    # which is increased at each new request. The id is send via
    # the dest_seq. For pre-request messages an offset is added
    # to recognize these meta-messages.
    #
    # We use an approach I call pre-request. The req channel sends
    # a pre-request to all repliers (on the same slot) asking whether
    # they want to handle a request. The content is 'req?' and the
    # dest_seq is the request-id + offset.
    #
    # The repliers collects and queues all pre-requests. It will then
    # send a reply to acknowledge the first received pre-request. The
    # content is 'req!' and dest_seq is again request-id + offset.
    #
    # The replier is now in a state of waiting for the actual request.
    # It will not acknowledge pre-requests, but keeps queing them.
    #
    # Upon receiving the acknowledge, the requester sends (directed
    # at only the first replier to acknowledge) the real request.
    # The content is the real request and dest_seq is the request-id.
    # Right after this, a pre-request cancel message is sent to all
    # repliers. The content is 'req-' and dest_seq is request-id + offset.
    #
    # When a replier receives a pre-request cancel message, it will
    # remove the pre-request from the list. If this cancels the
    # request it was currently waiting for, the replier will go back
    # to its default state, and acknowledge the first next pre-request
    # in the queue.
    #
    # When the replier answers a request, it will go back to its default
    # state, and acknowledge the first next pre-request in the queue.
    # The replier tries to answer as quickly to pre-requests as possible.
    #
    # On the request channel, a dictionary of request items is maintained.
    # Each item has an attribute specifying whether a replier has
    # acknowledged it (and which one).
    def __init__(self, context, slot_base):
        BaseChannel.__init__(self, context, slot_base, OBJECT)
        # Queue with pending requests
        self._request_items = {}
        # Timeout
        self._next_recheck_time = time.time() + 0.2
        # Counter
        self._request_counter = 0
        # The req channel is always in event driven mode
        self._run_mode = 1
        # Bind signals to process the events for this channel
        # Bind to "received" signal for quick response and a timer
        # so we can resend requests if we do not receive anything.
        self.received.bind(self._process_events_local)
        self._timer = yoton.events.Timer(0.5, False)
        self._timer.bind(self._process_events_local)
        self._timer.start()
    def _messaging_patterns(self):
        return "req-rep", "rep-req"
    def __getattr__(self, name):
        if name.startswith("_"):
            return object.__getattribute__(self, name)
        try:
            return object.__getattribute__(self, name)
        except AttributeError:
            def proxy_function(*args, **kwargs):
                return self._handle_request(name, *args, **kwargs)
            return proxy_function
    def _handle_request(self, name, *args, **kwargs):
        """_handle_request(request, callback, **kwargs)
        Post a request. This creates a Future instance and stores
        it. A message is send asking any repliers to respond.
        The actual request will be send when a reply to our pre-request
        is received. This all hapens in the yoton event loop.
        """
        # Create request object
        request = name, args, kwargs
        # Check and convert request message
        bb = self.message_to_bytes(request)
        # Get new request id
        request_id = self._request_counter = self._request_counter + 1
        # Create new item for this request and store under the request id
        item = Future(self, bb, request_id)
        self._request_items[request_id] = item
        # Send pre-request (ask repliers who want to reply to a request)
        item._send("req?")
        # Return the Future instance
        return item
    def _resend_requests(self):
        """_resend_requests()
        See if we should resend our older requests. Periodically calling
        this method enables doing a request while the replier is not yet
        attached to the network.
        This also allows the Future objects to cancel themselves if it
        takes too long.
        """
        for request_id in [key for key in self._request_items.keys()]:
            item = self._request_items[request_id]
            # Remove items that are really old
            if item.cancelled():
                self._request_items.pop(request_id)
            else:
                item._resend_if_necessary()
    def _recv_item(self):
        """_recv_item()
        Receive item. If a reply is send that is an acknowledgement
        of a replier that it wants to handle our request, the
        correpsonding request is send to that replier.
        This is a kind of mini-event loop thingy that should be
        called periodically to keep things going.
        """
        # Receive package
        package = self._recv(False)
        if not package:
            return
        # Get the package reply id and sequence number
        dest_id = package._dest_id
        request_id = package._dest_seq
        # Check dest_id
        if not dest_id:
            return  # We only want messages that are directed directly at us
        elif dest_id != self._context._id:
            return  # This should not happen; context should make sure
        if request_id > REQREP_SEQ_REF:
            # We received a reply to us asking who can handle the request.
            # Get item, send actual request. We set the replier to indicate
            # that this request is being handled, and we can any further
            # acknowledgements from other repliers.
            request_id -= REQREP_SEQ_REF
            item = self._request_items.get(request_id, None)
            if item and not item._replier:
                # Status now changes to "running" canceling is not possible
                ok = item.set_running_or_notify_cancel()
                if not ok:
                    return
                # Send actual request to specific replier
                try:
                    self._send(item._req, package._source_id, request_id)
                except IOError:
                    pass  # Channel closed, will auto-cancel at item._send()
                item._replier = package._source_id  # mark as being processed
                # Send pre-request-cancel message to everyone
                item._send("req-")
        elif request_id > 0:
            # We received a reply to an actual request
            # Get item, remove from queue, set reply, return
            item = self._request_items.pop(request_id, None)
            if item:
                item._rep = package._data
                return item
    def _process_events_local(self, dummy=None):
        """_process_events_local()
        Process events only for this object. Used by _handle_now().
        """
        # Check periodically if we should resend (or clean up) old requests
        if time.time() > self._next_recheck_time:
            self._resend_requests()
            self._next_recheck_time = time.time() + 0.1
        # Process all received messages
        while self.pending:
            item = self._recv_item()
            if item:
                reply = self.message_from_bytes(item._rep)
                if (
                    isinstance(reply, tuple)
                    and len(reply) == 2
                    and reply[0] == ERROR_OBJECT
                ):
                    item.set_exception(reply[1])
                else:
                    item.set_result(reply)
class RepChannel(BaseChannel):
    """RepChannel(context, slot_base)
    The reply part of the request/reply messaging pattern.
    A RepChannel instance receives request and sends the corresponding
    replies. The requests are send from a yoton.ReqChannel instance.
    This class adopts req/rep in a remote procedure call (RPC) scheme.
    To use a RepChannel, subclass this class and implement the methods
    that need to be available. The reply should be (a combination of)
    None, bool, int, float, string, list, tuple, dict.
    This channel needs to be set to event or thread mode to function
    (in the first case yoton events need to be processed too).
    To stop handling events again, use set_mode('off').
    Parameters
    ----------
    context : yoton.Context instance
        The context that this channel uses to send messages in a network.
    slot_base : string
        The base slot name. The channel appends an extension to indicate
        message type and messaging pattern to create the final slot name.
        The final slot is used to connect channels at different contexts
        in a network
    """
    def __init__(self, context, slot_base):
        BaseChannel.__init__(self, context, slot_base, OBJECT)
        # Pending pre-requests
        self._pre_requests = []
        # Current pre-request and time that it was acknowledged
        self._pre_request = None
        self._pre_request_time = 0
        # Create thread
        self._thread = ThreadForReqChannel(self)
        # Create timer (do not start)
        self._timer = yoton.events.Timer(2.0, False)
        self._timer.bind(self._process_events_local)
        # By default, the replier is off
        self._run_mode = 0
    def _messaging_patterns(self):
        return "rep-req", "req-rep"
    # Node that setters for normal and event_driven mode are specified in
    # channels_base.py
    def set_mode(self, mode):
        """set_mode(mode)
        Set the replier to its operating mode, or turn it off.
        Modes:
          * 0 or 'off': do not process requests
          * 1 or 'event': use the yoton event loop to process requests
          * 2 or 'thread': process requests in a separate thread
        """
        if isinstance(mode, basestring):
            mode = mode.lower()
        if mode in [0, "off"]:
            self._run_mode = 0
        elif mode in [1, "event", "event-driven"]:
            self._run_mode = 1
            self.received.bind(self._process_events_local)
            self._timer.start()
        elif mode in [2, "thread", "thread-driven"]:
            self._run_mode = 2
            if not self._thread.is_alive():
                self._thread.start()
        else:
            raise ValueError("Invalid mode for ReqChannel instance.")
    def _handle_request(self, message):
        """_handle_request(message)
        This method is called for each request, and should return
        a reply. The message contains the name of the method to call,
        this function calls that method.
        """
        # Get name and args
        name, args, kwargs = message
        # Get function
        if not hasattr(self, name):
            raise RuntimeError("Method '%s' not implemented." % name)
        else:
            func = getattr(self, name)
        # Call
        return func(*args, **kwargs)
    def _acknowledge_next_pre_request(self):
        # Cancel current pre-request ourselves if it takes too long.
        # Failsafe, only for if resetting by requester fails somehow.
        if time.time() - self._pre_request_time > 10.0:
            self._pre_request = None
        # Send any pending pre requests
        if self._pre_requests and not self._pre_request:
            # Set current pre-request and its ack time
            package = self._pre_requests.pop(0)
            self._pre_request = package
            self._pre_request_time = time.time()
            # Send acknowledgement
            msg = "req!".encode("utf-8")
            try:
                self._send(msg, package._source_id, package._dest_seq)
            except IOError:
                pass  # Channel closed, nothing we can do about that
            #
            # print 'ack', self._context.id,  package._dest_seq-REQREP_SEQ_REF
    def _replier_iteration(self, package):
        """_replier_iteration()
        Do one iteration: process one request.
        """
        # Get request id
        request_id = package._dest_seq
        if request_id > REQREP_SEQ_REF:
            # Pre-request stuff
            # Remove offset
            request_id -= REQREP_SEQ_REF
            # Get action and pre request id
            action = package._data.decode("utf-8")
            # Remove pre-request from pending requests in case of both actions:
            # Cancel pending pre-request, prevent stacking of the same request.
            for prereq in [prereq for prereq in self._pre_requests]:
                if (
                    package._source_id == prereq._source_id
                    and package._dest_seq == prereq._dest_seq
                ):
                    self._pre_requests.remove(prereq)
            if action == "req-":
                # Cancel current pre-request
                if (
                    self._pre_request
                    and package._source_id == self._pre_request._source_id
                    and package._dest_seq == self._pre_request._dest_seq
                ):
                    self._pre_request = None
            elif action == "req?":
                # New pre-request
                self._pre_requests.append(package)
        else:
            # We are asked to handle an actual request
            # We can reset the state
            self._pre_request = None
            # Get request
            request = self.message_from_bytes(package._data)
            # Get reply
            try:
                reply = self._handle_request(request)
            except Exception:
                reply = ERROR_OBJECT, getErrorMsg()
                print("yoton.RepChannel: error handling request:")
                print(reply[1])
            # Send reply
            if True:
                try:
                    bb = self.message_to_bytes(reply)
                    self._send(bb, package._source_id, request_id)
                except IOError:
                    pass  # Channel is closed
                except Exception:
                    # Probably wrong type of reply returned by handle_request()
                    print("Warning: request could not be send:")
                    print(getErrorMsg())
    def _process_events_local(self, dummy=None):
        """_process_events_local()
        Called when a message (or more) has been received.
        """
        # If closed, unregister from signal and stop the timer
        if self.closed or self._run_mode != 1:
            self.received.unbind(self._process_events_local)
            self._timer.stop()
        # Iterate while we receive data
        while True:
            package = self._recv(False)
            if package:
                self._replier_iteration(package)
                self._acknowledge_next_pre_request()
            else:
                # We always enter this the last time
                self._acknowledge_next_pre_request()
                return
    def echo(self, arg1, sleep=0.0):
        """echo(arg1, sleep=0.0)
        Default procedure that can be used for testing. It returns
        a tuple (first_arg, context_id)
        """
        time.sleep(sleep)
        return arg1, hex(self._context.id)
class ThreadForReqChannel(threading.Thread):
    """ThreadForReqChannel(channel)
    Thread to run a RepChannel in threaded mode.
    """
    def __init__(self, channel):
        threading.Thread.__init__(self)
        # Check channel
        if not isinstance(channel, RepChannel):
            raise ValueError("The given channel must be a REP channel.")
        # Store channel
        self._channel = channel
        # Make deamon
        self.setDaemon(True)
    def run(self):
        """run()
        The handler's main loop.
        """
        # Get ref to channel. Remove ref from instance
        channel = self._channel
        del self._channel
        while True:
            # Stop?
            if channel.closed or channel._run_mode != 2:
                break
            # Wait for data (blocking, look Rob, without spinlocks :)
            package = channel._recv(2.0)
            if package:
                channel._replier_iteration(package)
                channel._acknowledge_next_pre_request()
            else:
                channel._acknowledge_next_pre_request()
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module yoton.channels.channels_state
Defines the channel class for state.
"""
from yoton.misc import bytes
from yoton.channels import BaseChannel
class StateChannel(BaseChannel):
    """StateChannel(context, slot_base, message_type=yoton.TEXT)
    Channel class for the state messaging pattern. A state is synchronized
    over all state channels of the same slot. Each channel can
    send (i.e. set) the state and recv (i.e. get) the current state.
    Note however, that if two StateChannel instances set the state
    around the same time, due to the network delay, it is undefined
    which one sets the state the last.
    The context will automatically call this channel's send_last()
    method when a new context enters the network.
    The recv() call is always non-blocking and always returns the last
    received message: i.e. the current state.
    There are no limitations for this channel if events are not
    processed, except that the received signal is not emitted.
    Parameters
    ----------
    context : yoton.Context instance
        The context that this channel uses to send messages in a network.
    slot_base : string
        The base slot name. The channel appends an extension to indicate
        message type and messaging pattern to create the final slot name.
        The final slot is used to connect channels at different contexts
        in a network
    message_type : yoton.MessageType instance
        (default is yoton.TEXT)
        Object to convert messages to bytes and bytes to messages.
        Users can create their own message_type class to let channels
        any type of message they want.
    """
    def __init__(self, *args, **kwargs):
        BaseChannel.__init__(self, *args, **kwargs)
        # Variables to hold the current state. We use only the message
        # as a reference, so we dont need a lock.
        # The package is used to make _recv() function more or less,
        # and to be able to determine if a state was set (because the
        # message may be set to None)
        self._current_package = None
        self._current_message = self.message_from_bytes(bytes())
    def _messaging_patterns(self):
        return "state", "state"
    def send(self, message):
        """send(message)
        Set the state of this channel.
        The state-message is queued and send over the socket by the IO-thread.
        Zero-length messages are ignored.
        """
        # Send message only if it is different from the current state
        # set current_message by unpacking the send binary. This ensures
        # that if someone does this, things still go well:
        #    a = [1,2,3]
        #    status.send(a)
        #    a.append(4)
        #    status.send(a)
        if message != self._current_message:
            self._current_package = self._send(self.message_to_bytes(message))
            self._current_message =
self.message_from_bytes(self._current_package._data)
    def send_last(self):
        """send_last()
        Resend the last message.
        """
        if self._current_package is not None:
            self._send(self.message_to_bytes(self._current_message))
    def recv(self, block=False):
        """recv(block=False)
        Get the state of the channel. Always non-blocking. Returns the
        most up to date state.
        """
        return self._current_message
    def _recv_package(self, package):
        """_recv_package(package)
        Bypass queue and just store it in a variable.
        """
        self._current_message = self.message_from_bytes(package._data)
        self._current_package = package
        #
        self._maybe_emit_received()
    def _inject_package(self, package):
        """Non-blocking version of recv_package. Does the same."""
        self._current_message = self.message_from_bytes(package._data)
        self._current_package = package
        #
        self._maybe_emit_received()
    def _recv(self, block=None):
        """_recv(block=None)
        Returns the last received or send set package. The package
        may not reflect the current state.
        """
        return self._current_package
```

```
# -*- coding: utf-8 -*-
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
""" Module yoton.channels.message_types
Defines a few basic message_types for the channels. A MessageType object
defines how a message of that type should be converted to bytes and
vice versa.
The Packer and Unpacker classes for the ObjectMessageType are based on
the xdrrpc Python module by Rob Reilink and Windel Bouwman.
"""
import sys
import struct
from yoton.misc import bytes, basestring, long
# To decode P2k strings that are not unicode
if sys.__stdin__ and sys.__stdin__.encoding:
    STDINENC = sys.__stdin__.encoding
elif sys.stdin and sys.stdin.encoding:
    STDINENC = sys.stdin.encoding
else:
    STDINENC = "utf-8"
class MessageType(object):
    """MessageType()
    Instances of this class are used to convert messages to bytes and
    bytes to messages.
    Users can easily inherit from this class to make channels work for
    user specific message types. Three methods should be overloaded:
      * message_to_bytes()  - given a message, returns bytes
      * message_from_bytes() - given bytes, returns the message
      * message_type_name() - a string (for example 'text', 'array')
    The message_type_name() method is used by the channel to add an
    extension to the slot name, such that only channels of the same
    message type (well, with the same message type name) can connect.
    """
    def message_to_bytes(self, message):
        raise NotImplementedError()
    def message_from_bytes(self, bb):
        raise NotImplementedError()
    def message_type_name(self):
        raise NotImplementedError()
class BinaryMessageType(MessageType):
    """BinaryMessageType()
    To let channels handle binary messages.
    Available as yoton.BINARY.
    """
    def message_type_name(self):
        return "bin"
    def message_to_bytes(self, message):
        if not isinstance(message, bytes):
            raise ValueError("Binary channel requires byte messages.")
        return message
    def message_from_bytes(self, bb):
        return bb
class TextMessageType(MessageType):
    """BinaryMessageType()
    To let channels handle Unicode text messages.
    Available as yoton.TEXT.
    """
    def message_type_name(self):
        return "txt"
    def message_to_bytes(self, message):
        # Check
        if not isinstance(message, basestring):
            raise ValueError("Text channel requires string messages.")
        # If using py2k and the string is not unicode, make unicode first
        # by try encoding using UTF-8. When a piece of code stored
        # in a unicode string is executed, str objects are utf-8 encoded.
        # Otherwise they are encoded using __stdin__.encoding. In specific
        # cases, a non utf-8 encoded str might be succesfully encoded
        # using utf-8, but this is rare. Since I would not
        # know how to tell the encoding beforehand, we'll take our
        # chances... Note that in Pyzo (for which this package was created,
        # all executed code is unicode, so str instrances are always
        # utf-8 encoded.
        if isinstance(message, bytes):
            try:
                message = message.decode("utf-8")
            except UnicodeError:
                try:
                    message = message.decode(STDINENC)
                except UnicodeError:
                    # Probably not really a string then?
                    message = repr(message)
        # Encode and send
        return message.encode("utf-8")
    def message_from_bytes(self, bb):
        return bb.decode("utf-8")
class ObjectMessageType(MessageType):
    """ObjectMessageType()
    To let channels handle messages consisting of any of the following
    Python objects: None, bool, int, float, string, list, tuple, dict.
    Available as yoton.OBJECT.
    """
    def message_type_name(self):
        return "obj"
    def message_to_bytes(self, message):
        packer = Packer()
        packer.pack_object(message)
        return packer.get_buffer()
    def message_from_bytes(self, bb):
        if bb:
            unpacker = Unpacker(bb)
            return unpacker.unpack_object()
        else:
            return None
# Formats
_FMT_TYPE = "<B"
_FMT_BOOL = "<B"
_FMT_INT = "<q"
_FMT_FLOAT = "<d"
# Types
_TYPE_NONE = ord("n")
_TYPE_BOOL = ord("b")
_TYPE_INT = ord("i")
_TYPE_FLOAT = ord("f")
_TYPE_STRING = ord("s")
_TYPE_LIST = ord("l")
_TYPE_TUPLE = ord("t")
_TYPE_DICT = ord("d")
class Packer:
    # Note that while xdrlib uses StringIO/BytesIO, this approach using
    # a list is actually faster.
    def __init__(self):
        self._buf = []
    def get_buffer(self):
        return bytes().join(self._buf)
    def write(self, bb):
        self._buf.append(bb)
    def write_number(self, n):
        if n < 255:
            self.write(struct.pack("<B", n))
        else:
            self.write(struct.pack("<B", 255))
            self.write(struct.pack("<Q", n))
    def pack_object(self, object):
        if object is None:
            self.write(struct.pack(_FMT_TYPE, _TYPE_NONE))
        elif isinstance(object, bool):
            self.write(struct.pack(_FMT_TYPE, _TYPE_BOOL))
            self.write(struct.pack(_FMT_BOOL, object))
        elif isinstance(object, (int, long)):
            self.write(struct.pack(_FMT_TYPE, _TYPE_INT))
            self.write(struct.pack(_FMT_INT, object))
        elif isinstance(object, float):
            self.write(struct.pack(_FMT_TYPE, _TYPE_FLOAT))
            self.write(struct.pack(_FMT_FLOAT, object))
        elif isinstance(object, basestring):
            bb = object.encode("utf-8")
            self.write(struct.pack(_FMT_TYPE, _TYPE_STRING))
            self.write_number(len(bb))
            self.write(bb)
        elif isinstance(object, list):
            self.write(struct.pack(_FMT_TYPE, _TYPE_LIST))
            self.write_number(len(object))
            for value in object:
                self.pack_object(value)  # call recursive
        elif isinstance(object, tuple):
            self.write(struct.pack(_FMT_TYPE, _TYPE_TUPLE))
            self.write_number(len(object))
            for value in object:
                self.pack_object(value)  # call recursive
        elif isinstance(object, dict):
            self.write(struct.pack(_FMT_TYPE, _TYPE_DICT))
            self.write_number(len(object))
            # call recursive
            for key in object:
                self.pack_object(key)
                self.pack_object(object[key])
        else:
            raise ValueError("Unsupported type: %s" % repr(type(object)))
class Unpacker:
    def __init__(self, data):
        self._buf = data
        self._pos = 0
    def read(self, n):
        i1 = self._pos
        i2 = self._pos + n
        if i2 > len(self._buf):
            raise EOFError
        else:
            self._pos = i2
            return self._buf[i1:i2]
    def read_number(self):
        (n,) = struct.unpack("<B", self.read(1))
        if n == 255:
            (n,) = struct.unpack("<Q", self.read(8))
        return n
    def unpack(self, fmt, n):
        i1 = self._pos
        i2 = self._pos + n
        if i2 > len(self._buf):
            raise EOFError
        else:
            self._pos = i2
            data = self._buf[i1:i2]
            return struct.unpack(fmt, data)[0]
    def unpack_object(self):
        object_type = self.unpack(_FMT_TYPE, 1)
        if object_type == _TYPE_NONE:
            return None
        elif object_type == _TYPE_BOOL:
            return bool(self.unpack(_FMT_BOOL, 1))
        elif object_type == _TYPE_INT:
            return self.unpack(_FMT_INT, 8)
        elif object_type == _TYPE_FLOAT:
            return self.unpack(_FMT_FLOAT, 8)
        elif object_type == _TYPE_STRING:
            n = self.read_number()
            return self.read(n).decode("utf-8")
        elif object_type == _TYPE_LIST:
            object = []
            for i in range(self.read_number()):
                object.append(self.unpack_object())
            return object
        elif object_type == _TYPE_TUPLE:
            object = []
            for i in range(self.read_number()):
                object.append(self.unpack_object())
            return tuple(object)
        elif object_type == _TYPE_DICT:
            object = {}
            for i in range(self.read_number()):
                key = self.unpack_object()
                object[key] = self.unpack_object()
            return object
        else:
            raise ValueError("Unsupported type: %s" % repr(object_type))
# Define constants
TEXT = TextMessageType()
BINARY = BinaryMessageType()
OBJECT = ObjectMessageType()
if __name__ == "__main__":
    # Test
    s = {}
    s["foo"] = 3
    s["bar"] = 9
    s["empty"] = []
    s[(2, "aa", 3)] = ["pretty", ("nice", "eh"), 4]
    bb = OBJECT.message_to_bytes(s)
    s2 = OBJECT.message_from_bytes(bb)
    print(s)
    print(s2)
```

```
# -*- coding: utf-8 -*-
# flake8: noqa
# Copyright (C) 2013, the Pyzo development team
#
# Yoton is distributed under the terms of the 2-Clause BSD License.
# The full license can be found in 'license.txt'.
"""
The channel classes represent the mechanism for the user to send
messages into the network and receive messages from it. A channel
needs a context to function; the context represents a node in the
network.
"""
from yoton.channels.message_types import MessageType, TEXT, BINARY, OBJECT
from yoton.channels.channels_base import BaseChannel
from yoton.channels.channels_pubsub import PubChannel, SubChannel,
select_sub_channel
from yoton.channels.channels_reqrep import (
    ReqChannel,
    RepChannel,
    Future,
    TimeoutError,
    CancelledError,
)
from yoton.channels.channels_state import StateChannel
from yoton.channels.channels_file import FileWrapper
```

```
import os
def count_lines(filename):
    # Get text
    text = open(filename, "r").read()
    # Init counts: code, docstring, comments, whitespace
    count1, count2, count3, count4 = 0, 0, 0, 0
    inDocstring = False
    for line in text.splitlines():
        line = line.strip()
        if '"' * 3 in line:
            inDocstring = not inDocstring
        if not line:
            count4 += 1
        elif inDocstring:
            count2 += 1
        elif line.startswith("#") or line.startswith("%"):
            count3 += 1
        else:
            count1 += 1
    # Done
    return count1, count2, count3, count4
if __name__ == "__main__":
    # Get path of yoton
    path = os.path.dirname(os.path.abspath(__file__))
    path = os.path.split(path)[0]
    # Init files
    files = []
    # Get files in root
    for fname in os.listdir(path):
        files.append(fname)
    # Get files in channels
    for fname in os.listdir(os.path.join(path, "channels")):
        fname = os.path.join("channels", fname)
        files.append(fname)
    N1, N2, N3, N4 = 0, 0, 0, 0
    for fname in files:
        if not fname.endswith(".py"):
            continue
        n1, n2, n3, n4 = count_lines(os.path.join(path, fname))
        N1 += n1
        N2 += n2
        N3 += n3
        N4 += n4
        print("%i lines in %s" % (n1, fname))
    print("yoton has %i lines in its source" % (N1 + N2 + N3 + N4))
    print("yoton has %i lines of code" % N1)
    print("yoton has %i lines of docstring" % N2)
    print("yoton has %i comment lines" % N3)
    print("yoton has %i lines of whitespace" % N4)
```

```
# -*- coding: utf-8 -*-
# This example demponstrates simple req rep.
#
# This time in event driven mode. This example only works locally, as
# we cannot start two event loops :)
## ========== One end
import yoton
verbosity = 0
# Create a replier class by subclassing RepChannel
class Reducer(yoton.RepChannel):
    def reduce(self, item):
        if item == 2:
            raise ValueError("I do not like 2.")
        return item - 1
# Create a context and a rep channel
ct1 = yoton.Context(verbose=verbosity)
rep = Reducer(ct1, "reduce")
# Connect and turn duplicator on
ct1.bind("publichost:test")
rep.set_mode("event")
## ========== Other end
import yoton
import time
verbosity = 0
# Create a context and a req channel
ct2 = yoton.Context(verbose=verbosity)
req = yoton.ReqChannel(ct2, "reduce")
# Connect
ct2.connect("publichost:test")
# Create reply handler and bind it
def reply_handler(future):
    # Check error, cancelled, or get number
    if future.exception():
        # Calling result() would raise the exception, so lets just
        # print it and make up our own number
        print("oops: " + str(future.exception()))
        number = 1
    elif future.cancelled():
        print("oops: request was cancelled.")
    else:
        number = future.result()
    if number > 0:
        print("we now have %i." % number)
        time.sleep(0.5)
        new_future = req.reduce(number)
        new_future.add_done_callback(reply_handler)
    else:
        print("Done")
        yoton.stop_event_loop()
# Send first message
new_future = req.reduce(7)
new_future.add_done_callback(reply_handler)
# Enter event loop
yoton.start_event_loop()
```

```
# -*- coding: utf-8 -*-
# This example demponstrates simple pub sub.
#
# This time in event driven mode. This example only works locally, as
# we cannot start two event loops :)
## ========== One end
import yoton
verbosity = 0
# Create a context and a pub channel
ct1 = yoton.Context(verbose=verbosity)
pub = yoton.PubChannel(ct1, "foo")
# Connect and turn duplicator on
ct1.bind("publichost:test")
## ========== Other end
import yoton
verbosity = 0
# Create a context and a sub channel
ct2 = yoton.Context(verbose=verbosity)
sub = yoton.SubChannel(ct2, "foo")
# Connect, set channel to event driven mode
ct2.connect("publichost:test")
# Create message handler
def message_handler():
    message = sub.recv(False)
    if message:
        print(message)
        if message.lower() == "stop":
            yoton.stop_event_loop()
# Bind handler to a timer
timer = yoton.Timer(0.1, False)
timer.bind(message_handler)
timer.start()
# Send messages
yoton.call_later(pub.send, 8, "stop")
yoton.call_later(pub.send, 2, "2 seconds")
yoton.call_later(pub.send, 4, "4 seconds")
yoton.call_later(pub.send, 6, "almost done")
# Enter event loop
yoton.start_event_loop()
```

```
# -*- coding: utf-8 -*-
# This example illustrates a simple pub/sub pattern.
# This example can be run in one go or in two parts running in
# different processes.
## ========== One end
import yoton
verbosity = 0  # Python 2.4 can crash with verbosity on
# Create one context and a pub channel
ct1 = yoton.Context(verbose=verbosity)
pub = yoton.PubChannel(ct1, "chat")
# Connect
ct1.bind("publichost:test")
# Send
pub.send("hello world")
## ========== Other end
import yoton
verbosity = 0
# Create another context and a sub channel
ct2 = yoton.Context(verbose=verbosity)
sub = yoton.SubChannel(ct2, "chat")
# Connect
ct2.connect("publichost:test")
# Receive
print(sub.recv())
##
c1 = ct1.connections[0]
c2 = ct2.connections[0]
c1s = c1._sendingThread
c1r = c1._receivingThread
c2s = c2._sendingThread
c2r = c2._receivingThread
```

```
# -*- coding: utf-8 -*-
# This example creates three contexts that are connected in a row.
# A message is send at the first and received at the last.
import yoton
# Three contexts
verbosity = 2
c1 = yoton.Context(verbosity)
c2 = yoton.Context(verbosity)
c3 = yoton.Context(verbosity)
c4 = yoton.Context(verbosity)
# Connect in a row
addr = "localhost:whop"
c1.bind(addr + "+1")
c2.connect(addr + "+1")
c2.bind(addr + "+2")
c3.connect(addr + "+2")
c3.bind(addr + "+3")
c4.connect(addr + "+3")
# Create pub at first and sub at last
p = yoton.PubChannel(c1, "hop")
s = yoton.SubChannel(c4, "hop")
# Send a message.
p.send("hophophop")
print(s.recv())
```

```
# -*- coding: utf-8 -*-
# This example defines a message type and sends such a message over pub/sub.
# This example can be run in one go or in two parts running in
# different processes.
## ========== One end
import yoton
verbosity = 0
# Create custom message type. (should be defined at both ends)
class NumberMessageType(yoton.MessageType):
    def message_from_bytes(self, bb):
        return float(bb.decode("utf-8"))
    def message_to_bytes(self, number):
        return str(number).encode("utf-8")
    def message_type_name(self):
        return "num"
# Create context, a channel, and connect
ct1 = yoton.Context(verbose=verbosity)
pub = yoton.PubChannel(ct1, "numbers", NumberMessageType)
ct1.bind("publichost:test")
# Send a message
pub.send(42.9)
## ========== Other end
import yoton
verbosity = 0
# Create custom message type. (should be defined at both ends)
class NumberMessageType(yoton.MessageType):
    def message_from_bytes(self, bb):
        return float(bb)
    def message_to_bytes(self, number):
        return str(number).encode("utf-8")
    def message_type_name(self):
        return "num"
# Create a context, a channel, and connect
ct2 = yoton.Context(verbose=verbosity)
sub = yoton.SubChannel(ct2, "numbers", NumberMessageType)
ct2.connect("publichost:test")
# Duplicate a string and a number
print(sub.recv())
```

```
# -*- coding: utf-8 -*-
# This example demponstrates simple req rep.
# This example can be run in one go or in two parts running in
# different processes.
## ========== One end
import yoton
verbosity = 0
# Create a replier class by subclassing RepChannel
class Adder(yoton.RepChannel):
    def add(self, item1, item2):
        return item1 + item2
# Create a context and a rep channel
ct1 = yoton.Context(verbose=verbosity)
rep = Adder(ct1, "duplicate")
# Connect and turn duplicator on
ct1.bind("publichost:test")
rep.set_mode("thread")
## ========== Other end
import yoton
verbosity = 0
# Create a context and a req channel
ct2 = yoton.Context(verbose=verbosity)
req = yoton.ReqChannel(ct2, "duplicate")
# Connect
ct2.connect("publichost:test")
# Duplicate a string
print(req.add("foo", "bar").result(1))
print(req.add(3, 4).result(1))
```

```
# -*- coding: utf-8 -*-
import yoton
import time
# Create contexts
ct1 = yoton.Context()
ct2 = yoton.Context()
ct3 = yoton.Context()
ct4 = yoton.Context()
ct5 = yoton.Context()
# Connect
#            / ct3
# ct1 - ct2
#            \ ct4 - ct5
ct1.bind("localhost:split1")
ct2.connect("localhost:split1")
#
ct2.bind("localhost:split2")
ct3.connect("localhost:split2")
#
ct2.bind("localhost:split3")
ct4.connect("localhost:split3")
#
ct4.bind("localhost:split4")
ct5.connect("localhost:split4")
# Create channels
if True:
    pub1 = yoton.PubChannel(ct1, "splittest")
    sub1 = yoton.SubChannel(ct3, "splittest")
    sub2 = yoton.SubChannel(ct5, "splittest")
else:
    pub1 = yoton.StateChannel(ct1, "splittest")
    sub1 = yoton.StateChannel(ct3, "splittest")
    sub2 = yoton.StateChannel(ct5, "splittest")
# Go!
pub1.send("Hello you two!")
time.sleep(0.5)
print(sub1.recv())
print(sub2.recv())
```

```
# -*- coding: utf-8 -*-
## Connect two context
import yoton
import time
# Context 1
ct1 = yoton.Context()
pub1 = yoton.StateChannel(ct1, "state A")
pub1.send("READY")
# Context 2
ct2 = yoton.Context()
pub2 = yoton.StateChannel(ct2, "state A")
# Context 3
ct3 = yoton.Context()
sub1 = yoton.StateChannel(ct3, "state A")
# Connect
ct1.bind("localhost:test1")
ct2.connect("localhost:test1")
ct2.bind("localhost:test2")
ct3.connect("localhost:test2")
# Get status (but wait first)
time.sleep(0.3)
print(sub1.recv())
# New status
pub2.send("READY2")
time.sleep(0.3)
print(sub1.recv())
# And back to first
pub1.send("READY")
time.sleep(0.3)
print(sub1.recv())
## Now attach another context
# Upon connecting, the connecting context will issue a context-to-context
# message to indicate a new connection. All contexts will then call
# the send_last() methods of their state channels.
# Context 2
ct4 = yoton.Context()
sub2 = yoton.StateChannel(ct4, "state A")
# Connect
ct2.bind("localhost:test3")
ct4.connect("localhost:test3")
# Get status (but wait first)
time.sleep(0.3)
print(sub2.recv())
```

```
# Ask status again (simply gives last message)
print(sub2.recv())
## Using received signals
def on_new_state(channel):
    state = channel.recv()
    print("%i received state %s" % (id(channel), state))
    if state == "stop":
        yoton.stop_event_loop()
# Bind
sub1.received.bind(on_new_state)
sub2.received.bind(on_new_state)
# Have some calls made
yoton.call_later(pub1.send, 1.0, "hello")
yoton.call_later(pub1.send, 1.5, "there")
yoton.call_later(pub1.send, 2.0, "now")
yoton.call_later(pub1.send, 2.5, "stop")
# Go!
yoton.start_event_loop()
```

```
import yoton
import threading
import time
class Sender(threading.Thread):
    def run(self):
        for i in range(100):
            time.sleep(0.02)
            q.push("haha! " + repr(self))
        q.push("stop")
class Receiver(threading.Thread):
    def run(self):
        M = []
        L = []
        while True:
            L.append(len(q))
            m = q.pop(True)
            # time.sleep(0.02)
            if m == "stop":
                break
            else:
                M.append(m)
        self.M = M
        self.avSize = float(sum(L)) / len(L)
    def show(self):
        M = self.M
        nrs = {}
        for m in M:
            nr = m.split("-", 1)[1].split(",", 1)[0]
            if nr not in nrs:
                nrs[nr] = 0
            nrs[nr] += 1
        print("received %i messages" % len(M))
        print("average queue size was %1.2f" % self.avSize)
        for nr in nrs:
            print("  from %s received %i" % (nr, nrs[nr]))
q = yoton.misc.TinyPackageQueue(10, 1000)
S = []
for i in range(10):
    t = Sender()
    t.start()
    S.append(t)
R = []
for i in range(3):
    t = Receiver()
    t.start()
    R.append(t)
for r in R:
    r.join()
    r.show()
```

```
import yoton
verbosity = 0
def connect_test():
    c1 = yoton.Context(verbosity)
    c2 = yoton.Context(verbosity)
    c3 = yoton.Context(verbosity)
    c1.bind("localhost:test1")
    c2.connect("localhost:test1")
    c2.bind("localhost:test2")
    c3.connect("localhost:test2")
    c1.close()
    c2.close()
    c3.close()
for i in range(5):
    print("iter", i)
    connect_test()
```

```
""" Simple script to make a performance plot of the speed for sending
different package sizes.
"""
# Go up one directory and then import the codeeditor package
import os, sys
os.chdir("../..")
sys.path.insert(0, ".")
# Import yoton from there
import yoton
# Normal imports
import math
import time
import visvis as vv
## Run experiment with different message sizes
# Setup host
ct1 = yoton.Context()
ct1.bind("localhost:test")
c1 = yoton.PubChannel(ct1, "speedTest")
# Setup client
ct2 = yoton.SimpleSocket()
ct2.connect("localhost:test")
c2 = yoton.SubChannel(ct2, "speedTest")
# Init
minSize, maxSize = 2, 100 * 2**20
BPS = []
TPM = []
N = []
SIZE = []
# Loop
size = minSize
while size < maxSize:
    # Calculate amount
    n = int(200 * 2**20 / size)
    n = min(yoton.core.BUF_MAX_LEN, n)
    t0 = time.time()
    # Send messages
    message = "T" * int(size)
    for i in range(n):
        c1.send(message)
    ct1.flush(20.0)
    t1 = time.time()
    # In the mean while two threads are working their asses off to pop
    # the packages from one queue, send them over a socket, and push
    # them on another queue.
    # Receive messages
    for i in range(n):
        c2.recv()
    t2 = time.time()
    # Calculate speed
    etime = t2 - t0
    bps = n * size / etime  # bytes per second
    tpm = etime / n
    # Make strings
    bps_ = "%1.2f B/s" % bps
    size_ = "%i B" % size
    #
    D = {2**10: "KB", 2**20: "MB", 2**30: "GB"}
    for factor in D:
        if bps >= factor:
            bps_ = "%1.2f %s/s" % (bps / factor, D[factor])
        if size >= factor:
            size_ = "%1.2f %s" % (size / factor, D[factor])
    # Show result
    print("Sent %i messages of %s in %1.2f s: %s" % (n, size_, etime, bps_))
    # Store stuff
    N.append(n)
    SIZE.append(size)
    BPS.append(bps)
    TPM.append(tpm)
    # Prepare for next round
    size *= 1.9
## Visualize
def logticks10(unit="", factor=10):
    SIZE_TICKS = [factor**i for i in range(-50, 30, 1)]
    D = {}
    for i in SIZE_TICKS:
        il = math.log(i, factor)
        for j, c in reversed(zip([-6, -3, 1, 3], ["\\mu", "m", "", "K"])):
            jj = float(factor) ** j
            if i >= jj:
                i = "%1.0f %s%s" % (i / jj, c, unit)
                D[il] = i
                break
    return D
def logticks2(unit="", factor=2):
    SIZE_TICKS = [factor**i for i in range(-50, 30, 3)]
    D = {}
    for i in SIZE_TICKS:
        il = math.log(i, factor)
        for j, c in reversed(zip([1, 10, 20, 30], ["", "K", "M", "G"])):
            jj = float(factor) ** j
            if i >= jj:
                i = "%1.0f %s%s" % (i / jj, c, unit)
                D[il] = i
                break
    return D
SIZE_log = [math.log(i, 2) for i in SIZE]
BPS_log = [math.log(i, 2) for i in BPS]
TPM_log = [math.log(i, 10) for i in TPM]
fig = vv.figure(1)
vv.clf()
#
a1 = vv.subplot(211)
vv.plot(SIZE_log, BPS_log, ms=".")
vv.ylabel("speed [bytes/s]")
a1.axis.yTicks = logticks2()
#
a2 = vv.subplot(212)
vv.plot(SIZE_log, TPM_log, ms=".")  # 0.001 0.4
vv.ylabel("time per message [s]")
a2.axis.yTicks = logticks10("s")
for a in [a1, a2]:
    a.axis.xLabel = "message size"
    a.axis.showGrid = True
    a.axis.xTicks = logticks2("B")
# vv.screenshot('c:/almar/projects/yoton_performance.jpg', fig)
```

```
""" _utest.py
Unit tests for yoton.
"""
import sys, os, time
import unittest
# Go up one directory and then import the yoton package
os.chdir("..")
os.chdir("..")
sys.path.insert(0, ".")
# Import yoton from there
import yoton
from yoton.misc import long
class Tester(unittest.TestCase):
    """Perform a set of tests using three contexts and
    each of the six channels at each context.
    """
    def setUp(self):
        # Create contexts
        self._context1 = c1 = yoton.Context()
        self._context2 = c2 = yoton.Context()
        self._context3 = c3 = yoton.Context()
        # Create pub sub channels
        self._channel_pub1 = yoton.PubChannel(c1, "foo")
        self._channel_pub2 = yoton.PubChannel(c2, "foo")
        self._channel_pub3 = yoton.PubChannel(c3, "foo")
        #
        self._channel_sub1 = yoton.SubChannel(c1, "foo")
        self._channel_sub2 = yoton.SubChannel(c2, "foo")
        self._channel_sub3 = yoton.SubChannel(c3, "foo")
        # Create req rep channels
        self._channel_req1 = yoton.ReqChannel(c1, "bar")
        self._channel_req2 = yoton.ReqChannel(c2, "bar")
        self._channel_req3 = yoton.ReqChannel(c3, "bar")
        #
        self._channel_rep1 = yoton.RepChannel(c1, "bar")
        self._channel_rep2 = yoton.RepChannel(c2, "bar")
        self._channel_rep3 = yoton.RepChannel(c3, "bar")
        # Create state channels
        self._channel_state1 = yoton.StateChannel(c1, "spam")
        self._channel_state2 = yoton.StateChannel(c2, "spam")
        self._channel_state3 = yoton.StateChannel(c3, "spam")
    def tearDown(self):
        self._context1.close()
        self._context2.close()
        self._context3.close()
    def test_connecting(self):
        # Connect
        self._context1.bind("localhost:test1")
        self._context2.connect("localhost:test1")
        time.sleep(0.1)  # Give time for binding to finish
        # Test if connected
        self.assertEqual(self._context1.connection_count, 1)
        self.assertEqual(self._context2.connection_count, 1)
        self.assertEqual(self._context3.connection_count, 0)
        # Connect more
        self._context1.bind("localhost:test2")
        self._context3.connect("localhost:test2")
        time.sleep(0.1)  # Give time for binding to finish
        # Test if connected
        self.assertEqual(self._context1.connection_count, 2)
        self.assertEqual(self._context2.connection_count, 1)
        self.assertEqual(self._context3.connection_count, 1)
    def test_closing_channel(self):
        # Connect
        self._context1.bind("localhost:test")
        self._context2.connect("localhost:test")
        # Send data
        self._channel_pub1.send("hello")
        # Receive data
        self.assertEqual(self._channel_sub2.recv(), "hello")
        self.assertEqual(self._channel_sub2.recv(False), "")
        # Close channel
        self._channel_sub2.close()
        self.assertEqual(self._channel_sub2.recv(), "")
        self.assertTrue(self._channel_sub2.closed)
    def test_pub_sub(self):
        # also test hopping
        # Connect
        self._context1.bind("localhost:test1")
        self._context2.connect("localhost:test1")
        self._context2.bind("localhost:test2")
        self._context3.connect("localhost:test2")
        # Send messages
        self._channel_pub1.send("msg1")
        time.sleep(0.1)
        self._channel_pub2.send("msg2")
        time.sleep(0.1)
        # Receive msssages
        self.assertEqual(self._channel_sub2.recv(), "msg1")
        self.assertEqual(self._channel_sub3.recv(), "msg1")
        self.assertEqual(self._channel_sub3.recv(), "msg2")
    def test_pub_sub_select_channel(self):
        # Connect
        self._context1.bind("localhost:test1")
        self._context2.connect("localhost:test1")
        # Create channels
        pub1 = yoton.PubChannel(self._context1, "foo1")
        pub2 = yoton.PubChannel(self._context1, "foo2")
        #
        sub1 = yoton.SubChannel(self._context2, "foo1")
        sub2 = yoton.SubChannel(self._context2, "foo2")
        # Send a bunch of messages
        ii = [0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1]
        for i in range(len(ii)):
            pub = [pub1, pub2][ii[i]]
            pub.send(str(i))
        time.sleep(0.1)
        # Receive in right order
        count = 0
        while True:
            sub = yoton.select_sub_channel(sub1, sub2)
            if sub:
                i = int(sub.recv())
                self.assertEqual(i, count)
                count += 1
            else:
                break
        # Test count
        self.assertEqual(count, len(ii))
    def test_pubstate_substate(self):
        # also test hopping
        # Set status before connecting
        self._channel_state1.send("status1")
        # Connect
        self._context1.bind("localhost:test1")
        self._context2.connect("localhost:test1")
        self._context2.bind("localhost:test2")
        self._context3.connect("localhost:test2")
        time.sleep(0.1)
        # Receive msssages
        self.assertEqual(self._channel_state2.recv(), "status1")
        self.assertEqual(self._channel_state2.recv(), "status1")
        self.assertEqual(self._channel_state3.recv(), "status1")
        self.assertEqual(self._channel_state3.recv(), "status1")
        # Update status
        self._channel_state1.send("status2")
        time.sleep(0.1)
        # Receive msssages (recv multiple times should return last status)
        self.assertEqual(self._channel_state2.recv(), "status2")
        self.assertEqual(self._channel_state2.recv(), "status2")
        self.assertEqual(self._channel_state3.recv(), "status2")
        self.assertEqual(self._channel_state3.recv(), "status2")
    def test_req_rep1(self):
        # Connect
        self._context1.bind("localhost:test1")
        self._context2.connect("localhost:test1")
        self._context2.bind("localhost:test2")
        self._context3.connect("localhost:test2")
        # Turn on one replier
        self._channel_rep3.set_mode("thread")
        # Do requests
        foo, id1 = self._channel_req1.echo("foo").result(1)
        bar, id2 = self._channel_req1.echo("bar").result(1)
        # Check reply
        self.assertEqual(foo, "foo")
        self.assertEqual(bar, "bar")
        self.assertEqual(id1, hex(self._context3.id))
        self.assertEqual(id2, hex(self._context3.id))
    def test_req_rep2(self):
        # Test multiple requesters and multiple repliers
        # Connect
        if True:
            self._context1.bind("localhost:test1")
            self._context3.connect("localhost:test1")
            #
            self._context1.bind("localhost:test2")
            self._context2.connect("localhost:test2")
        else:
            freeNode = yoton.Context()
            #
            self._context1.bind("localhost:test1")
            freeNode.connect("localhost:test1")
            freeNode.bind("localhost:test1f")
            self._context3.connect("localhost:test1f")
            #
            self._context1.bind("localhost:test2")
            self._context2.connect("localhost:test2")
        time.sleep(0.1)
        # Turn on 2 requesters and 3 repliers
        self._channel_rep1.set_mode("thread")  # threaded because simulating
        self._channel_rep2.set_mode("thread")  # different processes
        self._channel_rep3.set_mode("thread")
        # Define and register reply handler
        def reply_handler(future):
            reply = future.result()
            reqnr = future.reqnr
            echo, id = reply
            if echo.lower() == "stop":
                yoton.stop_event_loop()
            else:
                self.assertEqual(echo[:3], "msg")
                contextnr = {
                    self._context1.id: 1,
                    self._context2.id: 2,
                    self._context3.id: 3,
                }[long(id, 16)]
                print(
                    "request %s from %i handled by context %i."
                    % (echo, reqnr, contextnr)
                )
        # Get echo functions
        echoFun1 = self._channel_req1.echo
        echoFun2 = self._channel_req2.echo
        # Send requests on req 1
        sleepTimes = [1, 0.1, 1, 0.6, 0.6, 0.6, 0.6, 0.6]
        for i in range(len(sleepTimes)):
            f = echoFun1("msg%i" % i, sleepTimes[i])
            f.add_done_callback(reply_handler)
            f.reqnr = 1
        # Send requests on req 2
        sleepTimes = [0.4, 0.4, 0.4, 0.4, 0.4]
        for i in range(len(sleepTimes)):
            f = echoFun2("msg%i" % i, sleepTimes[i])
            f.add_done_callback(reply_handler)
            f.reqnr = 2
        # Stop
        f = echoFun1("stop")
        f.add_done_callback(reply_handler)
        f.reqnr = 1
        # Enter event loop
        yoton.start_event_loop()
if __name__ == "__main__":
    suite = unittest.TestLoader().loadTestsFromTestCase(Tester)
    unittest.TextTestRunner(verbosity=2).run(suite)
    print("")
```

```
import numpy as np
import scipy as sp
# import scipy.linalg
import time
t0 = time.time()
a = np.random.normal(size=(1600, 1600))
sp.linalg.svd(a)
print(time.time() - t0)
# For channels.py, this results in pyzo closing the connection because
# the other side seems unresponsive.
```

```
import time
class Message(str):
    pass
# Create loads of messages of pure strings and Message instances with an attr
L1, L2 = [], []
for i in range(100000):
    text = "blabla" + str(i)
    tmp = Message(text)
    tmp._seq = i
    L1.append(text)
    L2.append(tmp)
# Test time to join
t0 = time.time()
for i in range(20):
    tmp = "".join(L1)
t1 = time.time() - t0
print(t1, "seconds for pure strings")
#
t0 = time.time()
for i in range(20):
    tmp = "".join(L2)
t2 = time.time() - t0
print(t2, "seconds for Message objects", 100 * (t2 - t1) / t1, "% slower")
# Test looking up
t0 = time.time()
for i in range(len(L2)):
    if L2[i]._seq > 9999999999999:
        break
print("i =", i, "  ", time.time() - t0, "seconds to check _seq attr")
t0 = time.time()
for i in range(0, len(L2), 10):
    if L2[i]._seq > 9999999999999:
        break
for i in range(i - 10, i):
    if L2[i]._seq > 9999999999999:
        break
print("i =", i, "  ", time.time() - t0, "seconds to check _seq attr")
```

```
"""
Small script to check the log generated by a test run (either frozen or source).
"""
import os
import sys
# Locate the log
logfilename = os.path.abspath(os.path.join(__file__, "..", "..", "log.txt"))
if not os.path.isfile(logfilename):
    raise RuntimeError(f"Pyzo log file not found in {logfilename}")
# Read and clear the log
with open(logfilename, "rt") as f:
    log = f.read()
os.remove(logfilename)
# Print the log
print("=" * 80)
print(log)
# Examine the results
if any(x in log.lower() for x in ["exception", "uncaught", "error"]):
    sys.exit("Errors detected during Pyzo test run :(")
elif not log.strip().endswith("Stopped"):
    sys.exit("Unclean stop for Pyzo test run :(")
else:
    print("==> Pyzo test run looks OK :)")
    sys.exit(0)
```

```
import sys
import pyzo
import subprocess
def test_api():
    assert pyzo.__version__
qt_libs = ["PySide", "PySide2", "PySide6", "PyQt4", "PyQt5", "PyQt6"]
code1 = """
import sys
import pyzo
print(list(sys.modules.keys()))
"""
def test_import1():
    x = subprocess.check_output([sys.executable, "-c", code1])
    modules = eval(x.decode())
    assert isinstance(modules, list)
    assert "sys" in modules
    assert "pyzo" in modules
    assert "pyzo.core" not in modules
    assert not any(qt_lib in modules for qt_lib in qt_libs)
    assert "pyzo.qt" not in modules
code2 = """
import sys
import pyzo
import pyzo.qt
print(list(sys.modules.keys()))
"""
def test_import2():
    x = subprocess.check_output([sys.executable, "-c", code2])
    modules = eval(x.decode())
    assert isinstance(modules, list)
    assert "sys" in modules
    assert "pyzo" in modules
    assert "pyzo.qt" in modules
    assert any(qt_lib in modules for qt_lib in qt_libs)
    assert "pyzo.core" not in modules
test_import1()
test_import2()
```