Coverage for **cylc/rose/rose.py** : 90%

172 statements | 154 run | 18 missing | 0 excluded

```python
# THIS FILE IS PART OF THE CYLC SUITE ENGINE.
# Copyright (C) NIWA & British Crown (Met Office) & Contributors.
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
"""Cylc support for reading and interpreting ``rose-suite.conf`` workflow
configuration files.
"""

import os
import re
import shlex

from pathlib import Path

from metomi.rose.config import (
    ConfigLoader, ConfigNodeDiff, ConfigDumper, ConfigNode
)
# from cylc.flow import LOG
from metomi.rose.config_processor import ConfigProcessError
from metomi.rose.env import env_var_process, UnboundEnvironmentVariableError
from metomi.rose import __version__ as ROSE_VERSION
from metomi.rose.resource import ResourceLocator

from cylc.flow.hostuserutil import get_host
from cylc.flow import LOG
from cylc.rose.jinja2_parser import Parser


class MultipleTemplatingEnginesError(Exception):
    ...


def get_rose_vars(dir_=None, opts=None):
    """Load Jinja2 Vars from rose-suite.conf in dir_

    Args:
        dir_(string or Pathlib.path object):
            Search for a ``rose-suite.conf`` file in this location.
        opts:
            Some sort of options object or string - To be used to allow CLI
            specification of optional configuaration.

    Returns:
        A dictionary of sections of rose-suite.conf.
        For each section either a dictionary or None is returned.
        E.g.
            {
                'env': {'MYVAR': 42},
                'empy:suite.rc': None,
                'jinja2:suite.rc': {
                    'myJinja2Var': {'yes': 'it is a dictionary!'}
                }
            }

    TODO:
        - Consider allowing ``[jinja2:flow.conf]`` as an alias for
          consistency with cylc.
    """
    config = {
        'env': {},
        'template_variables': {},
        'templating_detected': None
    }
    # Return a blank config dict if dir_ does not exist
    if not rose_config_exists(dir_):
        return config

    # Load the raw config tree
    config_tree = rose_config_tree_loader(dir_, opts)

    templating = None
    if (
        'jinja2:suite.rc' in config_tree.node.value and
        'empy:suite.rc' in config_tree.node.value
    ):
        raise MultipleTemplatingEnginesError(
            "You should not define both jinja2 and empy in the same "
            "configuration file."
        )
    elif 'jinja2:suite.rc' in config_tree.node.value:
        templating = 'jinja2'
    elif 'empy:suite.rc' in config_tree.node.value:
```

```python
            templating = 'empy'
        if templating:
            config['templating_detected'] = templating

        # Get Values for standard ROSE variables.
        rose_orig_host = get_host()
        rose_site = ResourceLocator().get_conf().get_value(['site'], '')

        # Create env section if it doesn't already exist.
        if 'env' not in config_tree.node.value:
            config_tree.node.set(['env'])

        # For each section add standard variables and process variables.
        for section in ['env', f'{templating}:suite.rc']:
            if section not in config_tree.node.value:
                continue

            # Add standard ROSE_VARIABLES
            config_tree.node[section].set(['ROSE_SITE'], rose_site)
            config_tree.node[section].set(['ROSE_VERSION'], ROSE_VERSION)
            config_tree.node[section].set(['ROSE_ORIG_HOST'], rose_orig_host)

            # Use env_var_process to process variables which may need expanding.
            for key, node in config_tree.node.value[section].value.items():
                try:
                    config_tree.node.value[
                        section
                    ].value[key].value = env_var_process(node.value)
                    if section == 'env':
                        os.environ[key] = node.value
                except UnboundEnvironmentVariableError as exc:
                    raise ConfigProcessError(['env', key], node.value, exc)

        # For each of the template language sections extract items to a simple
        # dict to be returned.
        if 'env' in config_tree.node.value:
            config['env'] = {
                item[0][1]: item[1].value for item in
                config_tree.node.value['env'].walk()
            }

        if f"{templating}:suite.rc" in config_tree.node.value:
            config['template_variables'] = {
                item[0][1]: item[1].value for item in
                config_tree.node.value[f"{templating}:suite.rc"].walk()
            }
        # Add the entire config to ROSE_SUITE_VARIABLES to allow for programatic
        # access.
        if templating is not None:
```

```python
        parser = Parser()
        for key, value in config['template_variables'].items():
            # The special variables are already Python variables.
            if key not in ['ROSE_ORIG_HOST', 'ROSE_VERSION', 'ROSE_SITE']:
                config['template_variables'][key] = parser.literal_eval(value)

    # Add ROSE_SUITE_VARIABLES to config of templating engines in use.
    if templating is not None:
        config['template_variables'][
            'ROSE_SUITE_VARIABLES'] = config['template_variables']

    # Add environment vars to the environment.
    for key, val in config['env'].items():
        os.environ[key] = val
    return config


def rose_fileinstall(dir_=None, opts=None, dest_root=None):
    """Call Rose Fileinstall.

    Args:
        dir_(string or pathlib.Path):
            Search for a ``rose-suite.conf`` file in this location.
        dest_root (string or pathlib.Path)

    """
    if not rose_config_exists(dir_):
        return False

    # Load the config tree
    config_tree = rose_config_tree_loader(dir_, opts)

    if any(['file' in i for i in config_tree.node.value]):

        # Carry out imports.
        from metomi.rose.config_processor import ConfigProcessorsManager
        from metomi.rose.popen import RosePopener
        from metomi.rose.reporter import Reporter
        from metomi.rose.fs_util import FileSystemUtil

        # Update config tree with install location
        # NOTE-TO-SELF: value=os.environ["CYLC_SUITE_RUN_DIR"]
        config_tree.node = config_tree.node.set(
            keys=["file-install-root"], value=dest_root
        )

        # Artificially set rose to verbose.
        # TODO - either use Cylc Log as event handler, or get Cylc Verbosity
        # settings to pass to Rose Reporter.
        event_handler = Reporter(3)
        fs_util = FileSystemUtil(event_handler)
        popen = RosePopener(event_handler)

        # Process files
        config_pm = ConfigProcessorsManager(event_handler, popen, fs_util)
        config_pm(config_tree, "file")

    return True


def rose_config_exists(dir_):
    """Does a directory contain a rose-suite config?

    Args:
        dir_(str or pathlib.Path object):
            location to test.

    Returns:

    """
    if dir_ is None:
        return False

    # Return None if rose-suite.conf do not exist.
    if isinstance(dir_, str):
        dir_ = Path(dir_)
    top_level_file = dir_ / 'rose-suite.conf'
    if not top_level_file.is_file():
        return False

    return True


def rose_config_tree_loader(dir_=None, opts=None):
    """Get a rose config tree from a given dir

    Args:
        dir_(string or Pathlib.path object):
            Search for a ``rose-suite.conf`` file in this location.
        opts:
            Some sort of options object or string - To be used to allow CLI
            specification of optional configuaration.
    Returns:
        A Rose ConfigTree object.
    """
    from metomi.rose.config_tree import ConfigTreeLoader

    opt_conf_keys = []
```

```python
    # get optional config key set as environment variable:
    opt_conf_keys_env = os.getenv("ROSE_SUITE_OPT_CONF_KEYS")
    if opt_conf_keys_env:
        opt_conf_keys += shlex.split(opt_conf_keys_env)
    # ... or as command line options
    if 'opt_conf_keys' in dir(opts) and opts.opt_conf_keys:
        opt_conf_keys += opts.opt_conf_keys

    # Optional definitions
    redefinitions = []
    if 'defines' in dir(opts) and opts.defines:
        redefinitions = opts.defines

    # Load the actual config tree
    config_tree = ConfigTreeLoader().load(
        str(dir_),
        'rose-suite.conf',
        opt_keys=opt_conf_keys,
        defines=redefinitions
    )

    return config_tree


def record_cylc_install_options(
    dest_root=None,
    opts=None,
    dir_=None,
):
    """Create/modify files recording Cylc install config options.

    Creates a new config based on CLI options and writes it to the workflow
    install location as ``rose-suite-cylc-install.conf``. If
    ``rose-suite-cylc-install.conf`` already exists over-writes changed items,
    except for ``!opts=`` which is merged and simplified.

    If ``!opts=`` have been changed these are appended to those that have
    been written in the installed ``rose-suite.conf``.

    Args:
        _ (pathlib.Path | or str):
            Not used in this function, but requried for consistent entry point.
        opts:
            Cylc option parser object - we want to extract the following
            values:
            - opt_conf_keys (list or str):
                Equivelent of ``rose suite-run --option KEY``
            - defines (list of str):
                Equivelent of ``rose suite-run --define KEY=VAL``
            - suite_defines (list of str):
                Equivelent of ``rose suite-run --define-suite KEY=VAL``
        dest_root (pathlib.Path | or str):
            Path to dump the rose-suite-cylc-conf

    Returns:
        cli_config - Config Node which has been dumped to
        ``rose-suite-cylc-install.conf``.
        rose_suite_conf['opts'] - Opts section of the config node dumped to
        installed ``rose-suite.conf``.
    """
    # Construct a path objects representing our target files.
    conf_filepath = Path(dest_root) / 'rose-suite-cylc-install.conf'
    rose_conf_filepath = Path(dest_root) / 'rose-suite.conf'
    dumper = ConfigDumper()
    loader = ConfigLoader()

    # Create a config based on command line options:
    cli_config = get_cli_opts_node(opts)

    # If file exists we need to merge with our new config, over-writing with
    # new items where there are duplicates.

    if conf_filepath.is_file():
        oldconfig = loader.load(str(conf_filepath))
        cli_config = merge_rose_cylc_suite_install_conf(oldconfig, cli_config)
    dumper(cli_config, str(conf_filepath))

    cli_config.comments = [
        ' This file records CLI Options.'
    ]
    dumper.dump(cli_config, str(conf_filepath))

    # Replace the opts section of the rose-suite.conf in the install location.
    # If we have not a rose-suite.conf but we use one of the Rose-style
    # options we still want to record those options.
    if not rose_conf_filepath.is_file():
        rose_conf_filepath.touch()
    rose_suite_conf = loader.load(str(rose_conf_filepath))
    rose_suite_conf = get_installed_rose_suite_conf_node(
        rose_suite_conf, cli_config
    )
    dumper(rose_suite_conf, rose_conf_filepath)
    return cli_config, rose_suite_conf['opts']


def merge_rose_cylc_suite_install_conf(old, new):
    """Merge old and new ``rose-suite-cylc-install.conf`` configs nodes.

```

```python
    Mostly this is straightforward, but special treatment is called for in
    the merger of opts.

    Args:
        old, new (ConfigNode):
            Old and new nodes.

    Returns:
        ConfigNode representing config to be written.

    Example:
        >>> from metomi.rose.config import ConfigNode;
        >>> old = ConfigNode({'opts': ConfigNode('a b c')})
        >>> new = ConfigNode({'opts': ConfigNode('c d e')})
        >>> merge_rose_cylc_suite_install_conf(old, new)['opts']
        {'value': 'a b c d e', 'state': '', 'comments': []}
    """
    new['opts'].value = simplify_opts_strings(
        old['opts'].value + ' ' + new['opts'].value
    )
    diff = ConfigNodeDiff()
    diff.set_from_configs(old, new)
    diff.delete_removed()
    old.add(diff)
    return old


def get_cli_opts_node(opts):
    """Create a node representing options set on the command line.

    Args:
        opts (CylcOptionParser object):
            Object with values from the command line.

    Returns:
        Cylc ConfigNode.

    Example:
        >>> from types import SimpleNamespace
        >>> opts = SimpleNamespace(
        ...     opt_confs='A B',
        ...     defines=["[env]FOO=BAR"],
        ...     define_suites=["QUX=BAZ"]
        ... )
        >>> node = get_cli_opts_node(opts)
        >>> node['opts']
        {'value': 'A B', 'state': '!', 'comments': []}
        >>> node['env']['FOO']
        {'value': 'BAR', 'state': '', 'comments': []}
        >>> node['jinja2:suite.rc']['QUX']
        {'value': 'BAZ', 'state': '', 'comments': []}
    """
    # Unpack options:
    opt_conf_keys = opts.opt_confs
    defines = opts.defines
    suite_defines = opts.define_suites

    # Construct new ouput based on optional Configs:
    newconfig = []
    newconfig = ConfigNode()

    # For each define determine whether it is an env or template define.
    for define in defines:
        match = re.match(
                (
                        r'^\[(?P<key1>.*)\](?P<state>!{0,2})'
                        r'(?P<key2>.*)\s*=\s*(?P<value>.*)'
                ),
                define
            ).groupdict()
        if match['key1'] == '' and match['state'] in ['!', '!!']:
            LOG.warning(
                'CLI opts set to ignored or trigger-ignored will be ignored.'
            )
        else:
            newconfig.set(
                keys=[match['key1'], match['key2']],
                value=match['value'],
                state=match['state']
            )

    # Write suite defines
    for define in suite_defines:
        # For now just assuming that we just support Jinja2 - after I've
        # Implemented the fully template-engine neutral template variables
        # section this should be a moot point.
        match = re.match(
            r'(?P<state>!{0,2})(?P<key>.*)\s*=\s*(?P<value>.*)', define
        ).groupdict()
        newconfig.set(
            keys=['jinja2:suite.rc', match['key']],
            value=match['value'],
            state=match['state']
        )

    # Specialised treatement of optional configs.
    if 'opts' not in newconfig:
        newconfig['opts'] = ConfigNode()
```

```
            newconfig['opts'].value = ''
        newconfig['opts'].value = merge_opts(newconfig, opt_conf_keys)
        newconfig['opts'].state = '!'

    return newconfig


def get_installed_rose_suite_conf_node(installed_conf, cli_conf):
    """Create a node with opts from installed rose-suite.conf + CLI opts.

    Args:
        installed_conf (ConfigNode):
            ConfigNode representing installed ``rose-suite.conf`` file.
        cli_conf (ConfigNode):
            ConfigNode representing CLI options.

    Returns:
        ConfigNode representing the final form of the installed
        ``rose-suite.conf`` file.

    Truth Table
        +-----------------------+---------------+------------------+
        |                       | New Opts      | New Opts ignored |
        +-----------------------+---------------+------------------+
        | No Existing opts      | New Opts      | Warning          |
        |                       |               | Nothing happens  |
        +-----------------------+---------------+------------------+
        | Existing opts         | Existing Opts | Warning          |
        |                       | + New Opts    | Nothing happens  |
        +-----------------------+---------------+------------------+
        | Existing opts ignored | New Opts      | Warning          |
        |                       |               | Nothing happens  |
        +-----------------------+---------------+------------------+

    Examples:
        >>> from metomi.rose.config import ConfigNode
        >>> foo = ConfigNode({'opts': ConfigNode('a b c')})
        >>> bar = ConfigNode({'opts': ConfigNode('c d e')})
        >>> result = get_installed_rose_suite_conf_node(foo, bar)
        >>> result['opts'].value
        'a b c c d e'
    """
    new_opts = ConfigNode()
    existing_opts_present = (
        'opts' in installed_conf and
        installed_conf['opts'] and
        installed_conf['opts'].state == ''
    )
    if (
```

```python
        existing_opts_present and
        'opts' in cli_conf and
        cli_conf['opts']
    ):
        new_opts.value = (
            f"{installed_conf['opts'].value}"
            f" {cli_conf['opts'].value}"
        )
    elif existing_opts_present:
        new_opts = installed_conf['opts']
    else:
        new_opts = cli_conf['opts']
    new_opts.comments = [(
        f' Config Options \'{new_opts.value}\' from CLI '
        'appended to options already in `rose-suite.conf`.'
    )]
    installed_conf['opts'] = new_opts
    # Do we actually want to do this?
    installed_conf['opts'].state = '!'
    return installed_conf


def merge_opts(config, opt_conf_keys):
    """Merge all options in specified order.

    Adds the keys for optional configs in order of increasing priority.
    Later items in the resultant string will over-ride earlier items.
    - Opts set using ``cylc install --defines "[]opts=A B C"``.
    - Opts set by setting ``ROSE_SUITE_OPT_CONF_KEYS="C D E"`` in environment.
    - Opts sey using ``cylc install --opt-conf-keys "E F G".

    In the example above the string returned would be "A B C D E F G".

    Args:
        config (ConfigNode):
            Config where opts has been added using ``--defines "[]opts=X"``.
        opt_conf_key (list | string):
            Options set using ``--opt-conf-keys "Y"`

    Returns:
        String containing opt conf keys sorted and with only the last of any
        duplicate.

    Examples:
        >>> from types import SimpleNamespace; conf = SimpleNamespace()
        >>> conf.value = 'aleph'; conf = {'opts': conf}

        Merge options from opt_conf_keys and defines.
        >>> merge_opts(conf, 'gimmel')
        'aleph gimmel'

        Merge options from defines and environment.
        >>> import os; os.environ['ROSE_SUITE_OPT_CONF_KEYS'] = 'bet'
        >>> merge_opts(conf, '')
        'aleph bet'

        Merge all three options.
        >>> merge_opts(conf, 'gimmel')
        'aleph bet gimmel'
    """
    all_opt_conf_keys = []
    if 'opts' in config:
        all_opt_conf_keys.append(config['opts'].value)
    if "ROSE_SUITE_OPT_CONF_KEYS" in os.environ:
        all_opt_conf_keys.append(os.environ["ROSE_SUITE_OPT_CONF_KEYS"])
    if opt_conf_keys and isinstance(opt_conf_keys, str):
        all_opt_conf_keys.append(opt_conf_keys)
    if opt_conf_keys and isinstance(opt_conf_keys, list):
        all_opt_conf_keys += opt_conf_keys
    return simplify_opts_strings(' '.join(all_opt_conf_keys))


def simplify_opts_strings(opts):
    """Merge Opts strings:

    Rules:
        - Items in new come after items in old.
        - Items in new are removed from old.
        - Otherwise order is preserved.

    Args:
        opts (str):
            a string containing a space delimeted list of options.
    Returns (str):
        A string which acts as a space delimeted list.

    Examples:
        >>> simplify_opts_strings('a b c')
        'a b c'
        >>> simplify_opts_strings('a b b')
        'a b'
        >>> simplify_opts_strings('a b a')
        'b a'
        >>> simplify_opts_strings('a b c d b')
        'a c d b'
        >>> simplify_opts_strings('a b c b d')
        'a c b d'
        >>> simplify_opts_strings('a b a b a a b b b c a b hello')
        'c a b hello'
    """

    seen_once = []
    for index, item in enumerate(reversed(opts.split())):
        if item not in seen_once:
            seen_once.append(item)

    return ' '.join(reversed(seen_once))
```