CBSE Computer Science Final Project

- Name : Aahnik Daw
- Class : XII-E
- School : Kalyani Public School, Barasat, West Bengal
- Roll :
- GitHub Repository : https://github.com/aahnik/cbse-xii-cs-proj/tree/main/project
- PyPI Release : https://pypi.org/project/marksman/
- API Reference (Site generated from docstrings) : https://aahnik.github.io/cbse-xii-cs-proj/marksman/

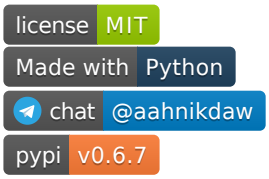
Scan this QR Code to visit the page online

README.md

*(NOTE : This markdown file has multiple GIFs. Please view the **online version** for best experience.)*

marksman

CLI Tool to manage marks of students efficiently.



Features

- **CRUD** (create / read / update / delete)
- **Email** (email results to students)
- **Visualize** (visualize the data)
- **Utils** (fill dummy data / import from csv / export csv)

Getting Started

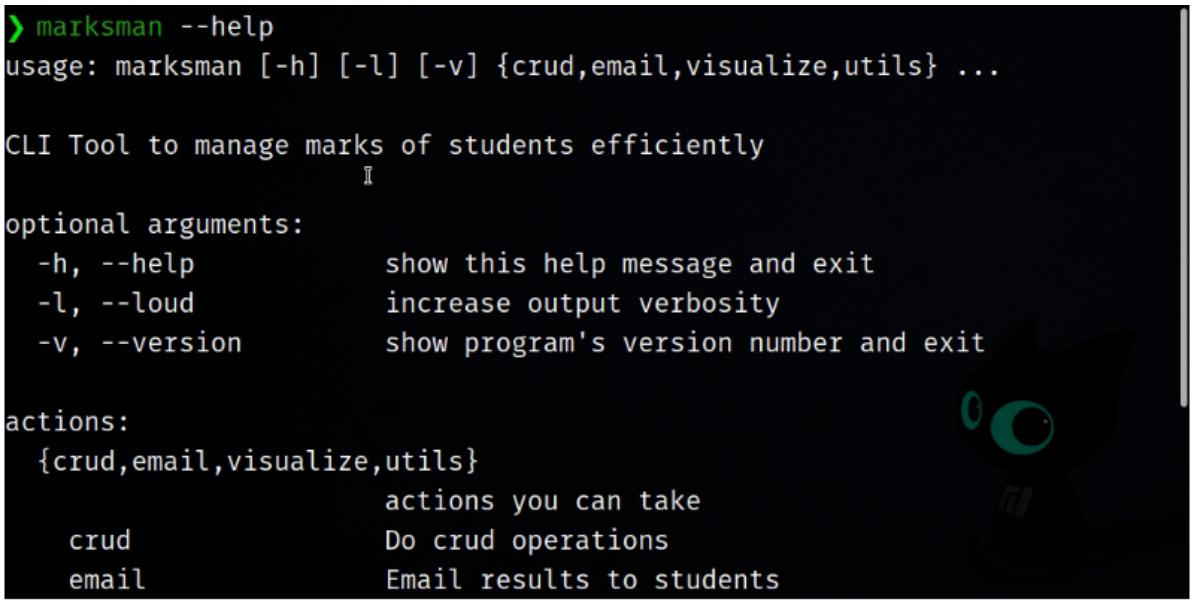
You must have Python 3.9 installed on your system.

Install easily via `pip`, Python's official Package Manager.

```
pip install marksman
```

> Note: Linux and Mac users might have to use `pip3` to invoke pip

Please read this if you are new to command-line-interfaces.



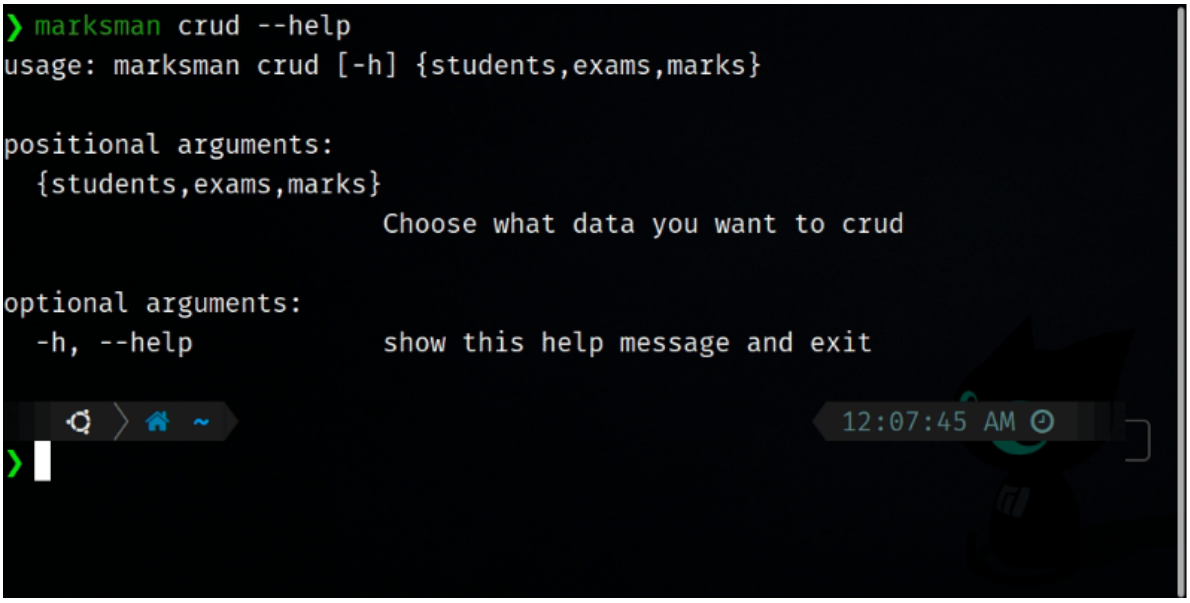
> **Tip**: You can use the alias `mm` instead of typing the long `marksman`. Its already set for you when you install.

CRUD

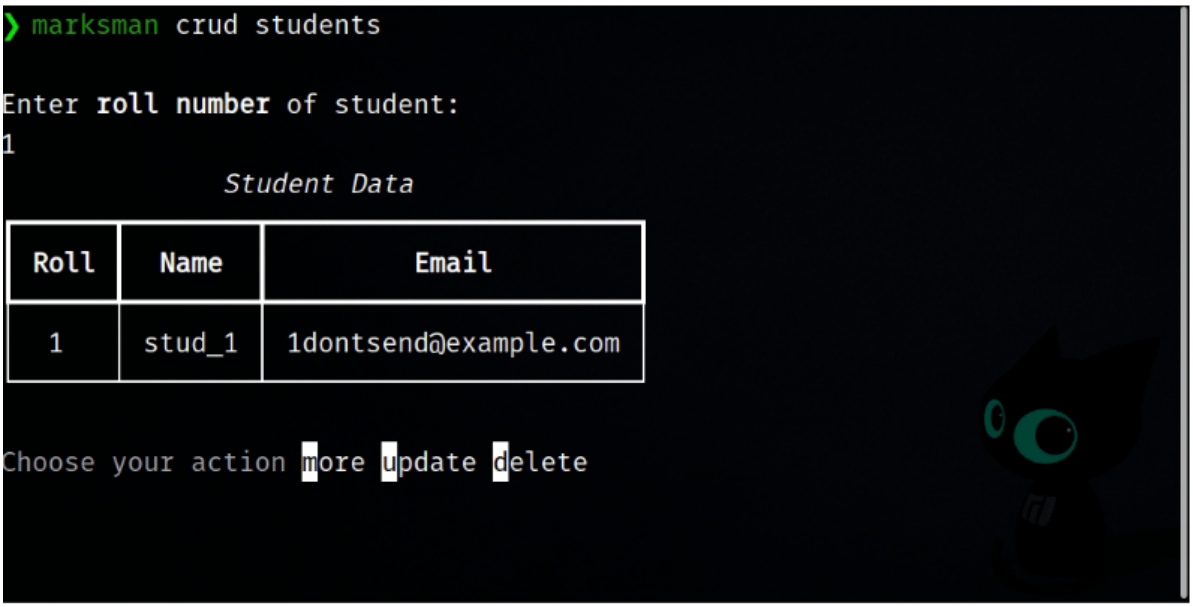
CRUD stands for Create,Read,Update,Delete. These are the fundamental operations we can perform on a database. `marksman` supports all these operations, through its CLI.

You can CRUD the students, exams and marks entries using `marksman`.

To use the CRUD action, first read its usage.



Suppose you want to CRUD students, here is an example.



If you have noticed the above gif carefully, you can observe that when you run the `crud` action with `students` as the option, you are first asked the roll number of the student. If a student with that roll exists, you are shown its details. You also have further options to see more data or update or delete the student.

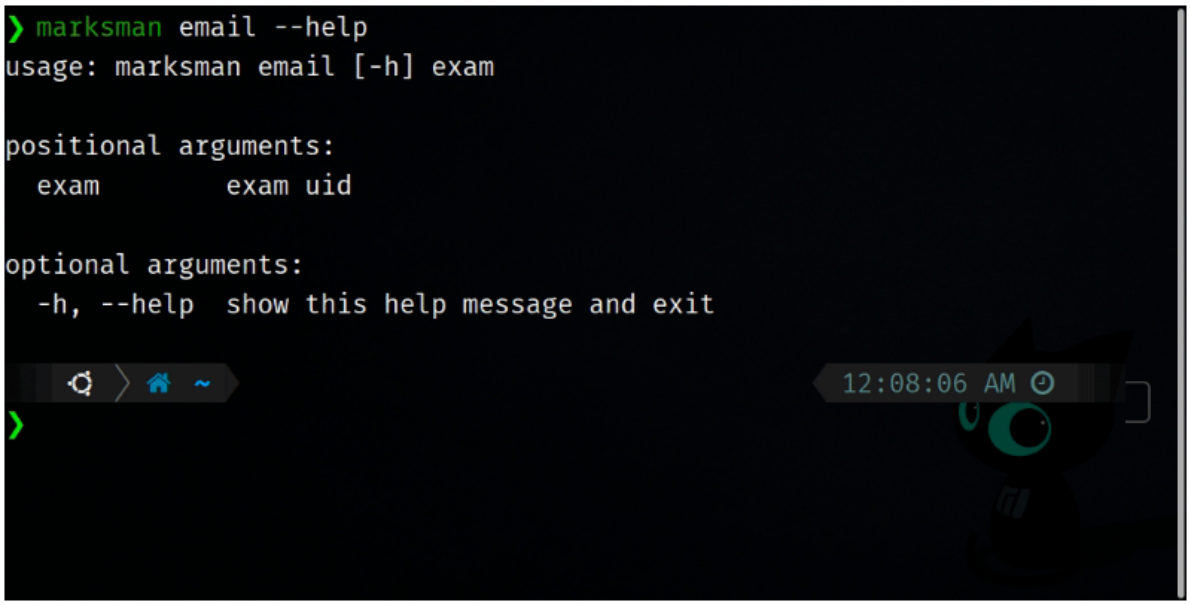
If the student, with the roll number you entered does not exist, then you will be given the choice to create the student.

If you don't enter any roll number, then `marksman` will display all students in the database and exit.

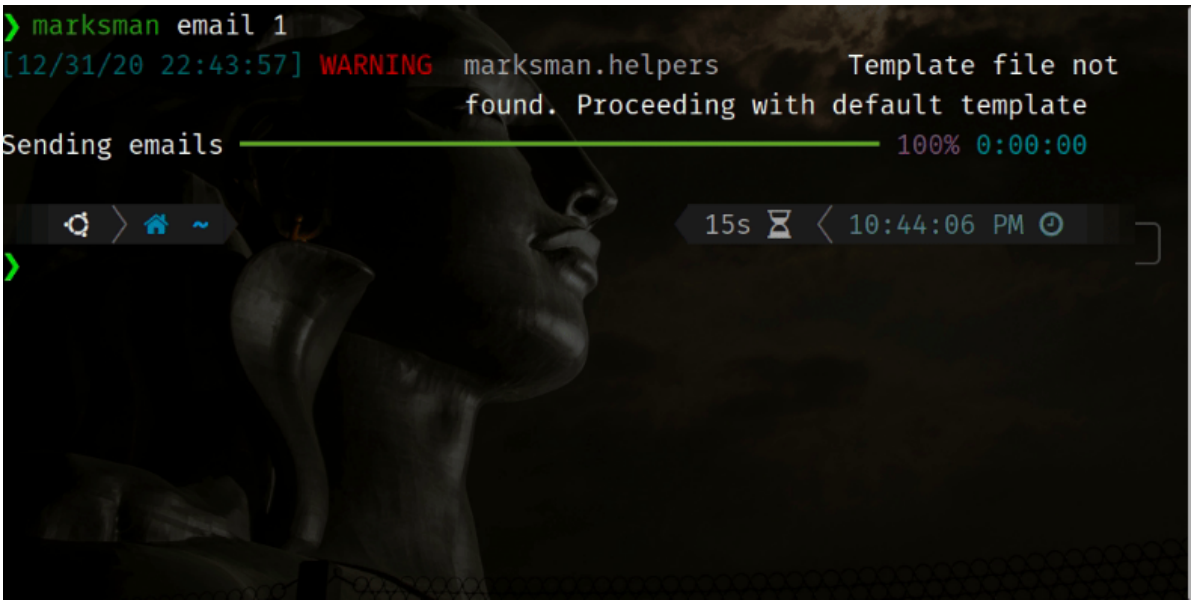
In the same fashion you can also `crud` exams and marks entries. See more examples of crud in action.

Email

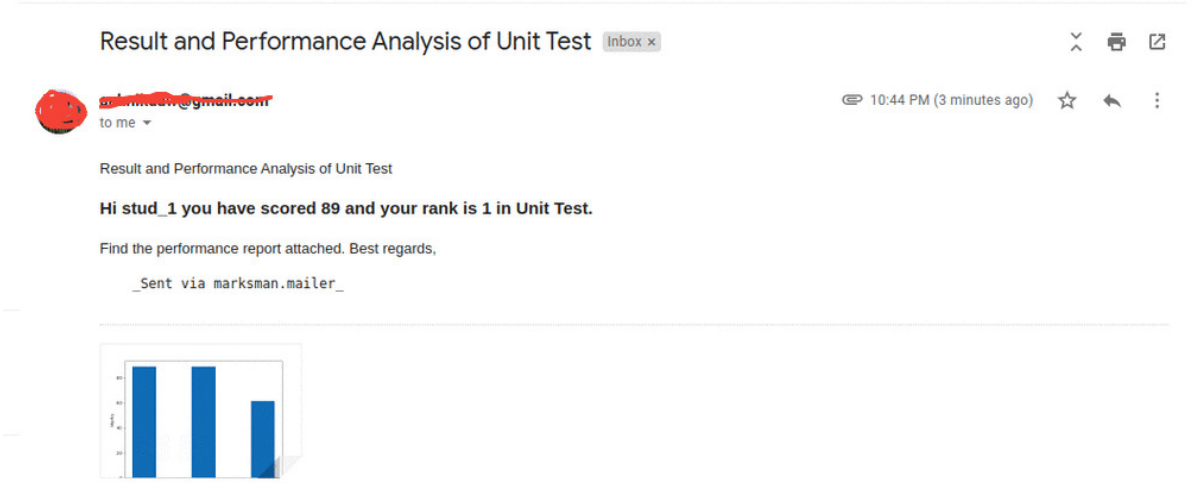
You can run `marksman email --help` to see how to use the email action. To configure settings for your emailing, read Configuration.



Now you have seen, how to use the `email` action, below is an example usage.



The result is this:



You can also use a custom template for the email message. If no template is found the default will be used.

For using a custom template, create a file called `marksman_email_template.md` and put in the directory from which you are running `marksman`.

You can insert variables in your template like this `::var::`.

Here is a sample template.

```
Hi ::name::, you have scored ::marks:: in ::exam::.

Highest marks is ::highest::
Average marks is ::average::
Your rank is ::rank::

Please find the attached graph of your performance analysis.

Best Regards,
::inst::

```

The list of supported variables are as follows:

List of variables passed to the template:

1. name
2. roll
3. email
4. exam
5. marks
6. rank
7. highest
8. average
9. inst (name of institute)

As this template is in a markdown file, you can use any valid markdown formatting.

Visualize

Data visualization is essential in taking important decisions. We can have a lot of data, but if we cannot make meaningful conclusions from it, then it is useless.

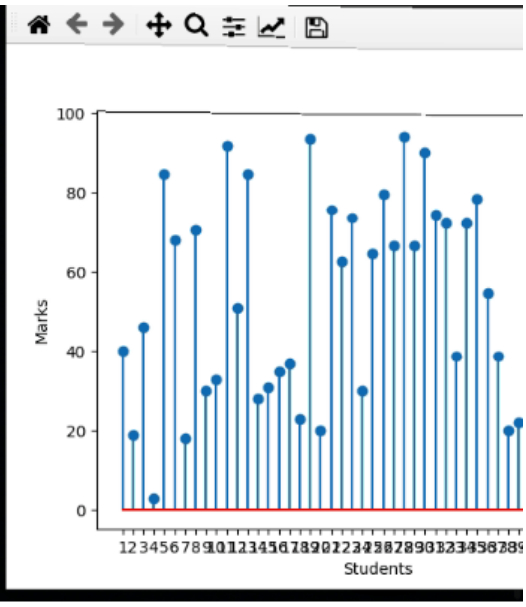
In `marksman` you have the data of marks of all students. `marksman` allows you to easily visualize the performance of the entire batch or of an individual student using simple graphs.

You can learn to use the `visualize` action by using the `--help` flag.

```
❯ marksman visualize --help
usage: marksman visualize [-h] [--r ROLL] exam

positional arguments:
  exam        exam uid

optional arguments:
  -h, --help  show this help message and exit
  --r ROLL    roll number of student (default=0 for all)

                                                    12:08:46 AM
❯
```

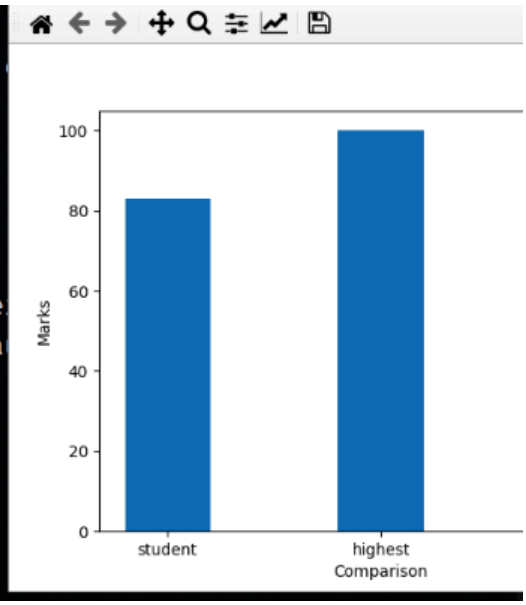
Let us see an example, where the teacher can visualize the results of an exam for the entire batch.

```
❯ marksman visualize 1
```



You can also visualize the result of one particular student in that exam. Just use the `--r` flag to supply the roll number of the student to `marksman`. See the example below for clarity.

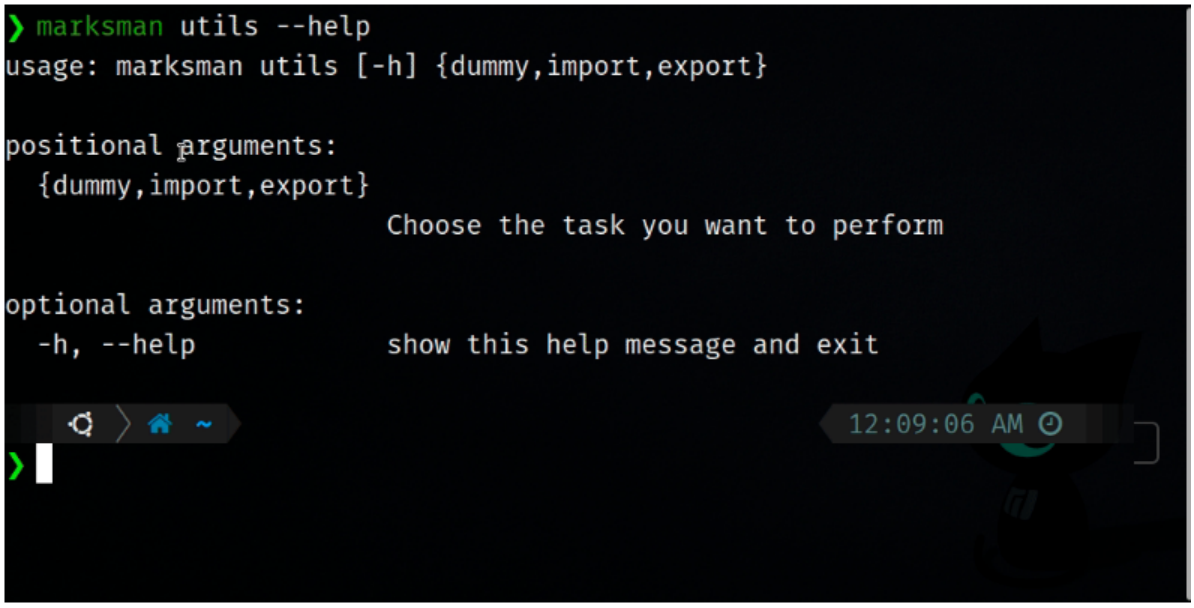
```
❯ marksman visualize --help
usage: marksman visualize [-h] [--r ROLL]

positional arguments:
  exam        exam uid

optional arguments:
  -h, --help  show this help message and e
  --r ROLL    roll number of student (defa
❯ marksman visualize 3 --r 1
```



Utils

`marksman` provides certain additional utilities to make your life easy. If you want to experiment with `marksman`, without actually putting any data your self, then `marksman` can help you by filling some dummy data. You can also import or export your data from or to CSV file formats.

To learn about the `utils` action, use the `--help` flag.

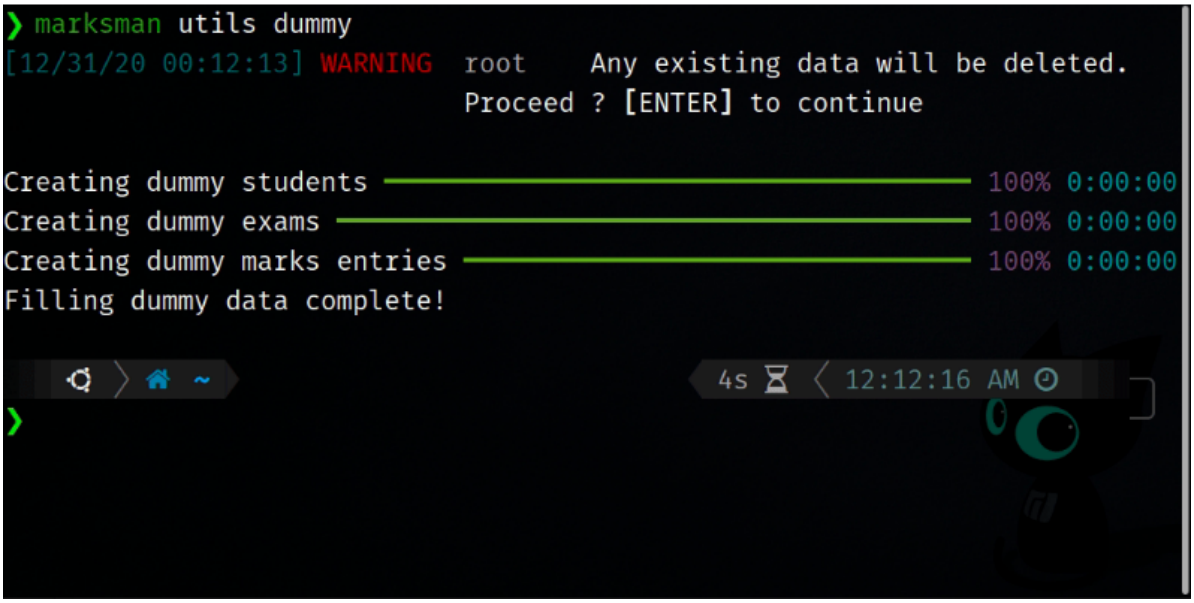


As you can see that `utils` can accept either of the three arguments:

1. `dummy` : fill the database with fake students, exams and marks for testing purposes.
2. `import` : insert data into the database from CSV files (of specific format)
3. `export` : write existing data to CSV files

Lets see some examples, for full clarity.

Filling dummy data is a piece of cake. Just run `marksman utils dummy`



Now when it comes to importing data from CSV files, it must be in a specific format.

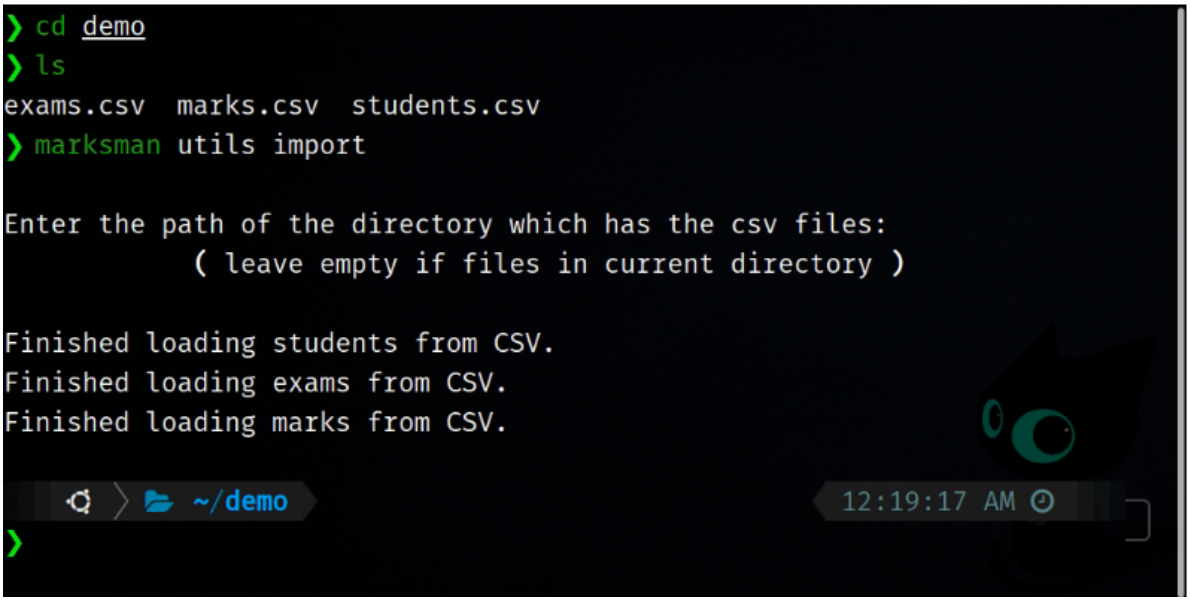
Here is a brief description of the convention that is supported by `marksman`:

Data	Filename	CSV Headers
Students	`students.csv`	`roll,name,email`
Exams	`exams.csv`	`uid,name`
Marks	`marks.csv`	`student,exam,marks`

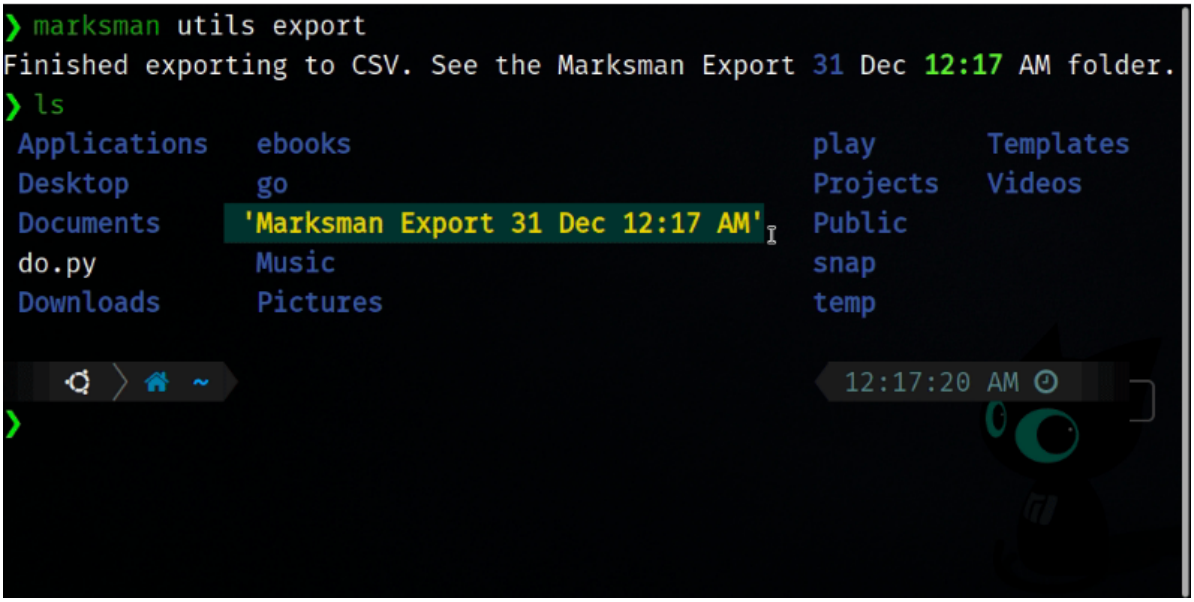
All three files must be kept in the same folder. For the CSV file for marks, `student` column should contain the roll, and `exam` the uid.

It is not compulsory to have all the three files for the `import`. Marksman will take whatever data you give it.

See the following example for understanding better.



You can use the `export` option to write the data in the database to `CSV` files.
The exported files will have the same format as shown above.



Configuration

You can configure marksman by using certain environment variables.

`marksman` loads the environment variables from the system's environment and from two special files:

1. the `.env` file in `~/.marksman` (global settings for `marksman`)
2. the `.env` file in the current directory from which you are running `marksman` (local settings for `marksman`)

The local settings will override the global settings.

> **Note:** `~` means the user directory. It is different for Unix and Windows.
> Windows 10: `~` expands to `<root>\Users\<username>`
> For Linux: `~` expands to `/home/<username>`
> The `~` symbol can be used directly in the terminal, let your OS expand it for you.

List of environment variables supported:

Variable	Description	Default
`marksman_db`	the path to store the `.db` file	`~/.marksman/database.db`
`marksman_sender`	email address to be used to send emails	
`marksman_auth`	password to login to sender's email account	
`marksman_smtp_host`	url of SMTP host server	`smtp.gmail.com`
`marksman_smtp_port`	SMTP port for sending emails	`587`
`marksman_inst`	name of institute which is sending emails	

You can write these settings in the `.env` file like this:

```
marksman_sender=meet.aahnik@gmail.com
marksman_auth=dummypass
```

If you do not provide a value, the default value will be used.

If a value is required, `marksman` will prompt you to to enter it during program execution. `marksman` will also ask you if you want to save it. If you say yes, `marksman` will save that setting in the global scope (`~/.marksman/.env` file). In this way, `marksman` can exempt you from manually editing the configs. You can always manually edit them when you want.

API Reference

As `marksman` is written purely in python, it can easily be imported and extended by other Python programs.

Read the full API Reference which is published from the docstrings.

Acknowledgements

I am extremely grateful to our Computer Science teacher Sir Mr. Biswarup Sarkar and our Principal Mam Mrs. Rupa Dey for providing us with excellent guidance and oppurtunities.

I am also grateful to the developers of the Python Language and the developers of other libraries on which my project is dependant.

I would like to extend my thanks to my friends and online communities like Stack Overflow and GitHub, which has provided me with fantastic resources.

Aahnik Daw

You can find some extra documentation at :

- https://github.com/aahnik/cbse-xii-cs-proj/blob/main/project/docs/cli_for_beginners.md#command-line-for-beginners
- https://github.com/aahnik/cbse-xii-cs-proj/blob/main/project/docs/more_usage_examples.md#examples

marksman

`__init__.py`

```
''' A command line utitility to manage marks of students.
Perform CRUD, email results or visualize data.
Also supports import and export from and to CSV files.


Source Code: https://github.com/aahnik/cbse-xii-cs-proj/tree/main/project
PyPI: https://pypi.org/project/marksman/
'''

__version__ = "0.6.7"

```

app.py

```python
''' All the handlers for actions the app supports are registered in this module
'''

from argparse import Namespace
import logging
from smtplib import SMTP, SMTPAuthenticationError
from sqlite3 import Cursor

from rich.console import Console

from marksman.db import DbModelz, create_tables
from marksman.validators import get_email, get_pos_int, get_str, roll, uid
from marksman.models import Models
from marksman.helpers import handle_choice, configure_email
from marksman.mailer import Mailer
from marksman.plot import analyse_exam, plot_student_performance, plot_batch_performance
from marksman.utils import ImportExport, fill_dummy

logger = logging.getLogger(__name__)

console = Console()


def crud_handler(args: Namespace, cursor: Cursor) -> None:
    ''' Handle the crud action

    Args:
        args (Namespace): the arguments provided at command line
        cursor (Cursor): sqlite3 cursor object
    '''
    logger.info(f'Called crud handler with {args.what}')

    pks_fns = {
        'students': {'roll': roll},
        'exams': {'uid': uid},
        'marks': {'student': roll, 'exam': uid}
    }

    values_fns = {
        'students': {'name': get_str, 'email': get_email},
        'exams': {'name': get_str},
        'marks': {'marks': get_pos_int}
    }

    db_modelz = DbModelz(args.what, cursor)
    pks_fn = pks_fns.get(args.what)
    values_fn = values_fns.get(args.what)

    model = Models(db_modelz=db_modelz, pks_fn=pks_fn, values_fn=values_fn)
    obj = model.object

    if not obj:
```

```
        logger.warning('Object does not exist')
        handle_choice({'create': model.create})
    else:
        model.display()

        handle_choice({'more': model.show_related,
                       'update': model.update, 'delete': model.delete})


def email_handler(args: Namespace, cursor: Cursor) -> None:
    ''' Handles the email action

    Args:
        args (Namespace): the arguments provided at command line
        cursor (Cursor): sqlite3 cursor object
    '''

    logger.info(f'Called email handler with {args.exam}')
    sender_email, sender_auth, smtp_host, smtp_port, inst_name =
configure_email()

    try:
        server = SMTP(host=smtp_host, port=smtp_port)
        server.connect(host=smtp_host, port=smtp_port)
    except Exception as err:
        logger.warning('Could not connect to SMTP server.')
        logger.exception(err)
        return
    else:
        logger.info('Connected to SMTP server')

    server.ehlo()
    server.starttls()
    server.ehlo()

    try:
        server.login(user=sender_email, password=sender_auth)
    except SMTPAuthenticationError as err:
        logger.warning(
            'Could not login to SMTP server using credentials you
provided')
        console.print(
            f'''\nMost probably you gave incorrect email and password!
            Error Code {err.smtp_code}\n{err.smtp_error}\n''')
        return

    mailer = Mailer(server=server, cursor=cursor,
                    sender=sender_email, inst=inst_name,
exam_uid=args.exam)

    mailer.mail_all_students()

    server.quit()
    logger.info('Disconnected from SMTP server')


def visualization_handler(args: Namespace, cursor: Cursor) -> None:
    ''' Handles the visualize action
```

```
    Args:
        args (Namespace): the arguments provided at command line
        cursor (Cursor): sqlite3 cursor object
    '''

    logger.info(f'Called vis handler with {args.exam} and {args.r}')
    if args.r == 0:
        # plot performance of batch
        plot_batch_performance(cursor, args.exam)
    elif args.r > 0:
        # plot performance of specific student
        plot_student_performance(
            cursor, args.r, args.exam, analyse_exam(cursor, args.exam))
    else:
        logger.warning('Roll number must be greater than 0')


def utils_handler(args: Namespace, cursor: Cursor) -> None:
    ''' Handles the utils action

    Args:
        args (Namespace): the arguments provided at command line
        cursor (Cursor): sqlite3 cursor object
    '''

    logger.info(f'Called utils with {args.task}')
    students = DbModelz('students', cursor)
    exams = DbModelz('exams', cursor)
    marks = DbModelz('marks', cursor)

    if args.task == 'dummy':
        logging.warning(
            'Any existing data will be deleted. Proceed ? [ENTER] to
continue')
        input()
        create_tables(cursor)
        fill_dummy(students, exams, marks)
    if args.task in ['import', 'export']:
        imp_exp = ImportExport(students, exams, marks)
        imp_exp.do_apt(args.task)

```

cli.py

```
''' Command line entry point for marksman '''

import os
import sys
import signal
from argparse import RawTextHelpFormatter, Namespace, ArgumentParser
import logging
import sqlite3

from rich.logging import RichHandler

from marksman import __version__
from marksman.db import create_tables, foreign_key_constraint
from marksman.helpers import ensure_parent
from marksman.settings import DB_PATH, LOUD, SHOW_PATH
from marksman.app import crud_handler, email_handler,
visualization_handler, utils_handler


logger = logging.getLogger(__name__)


def parse_commands() -> Namespace:
    ''' Parse the commands passed to marksman

    Returns:
        Namespace: [description]
    '''

    main_parser = ArgumentParser(
        description='CLI Tool to manage marks of students efficiently',
        epilog='''For tutorials and documentation visit
https://pypi.org/project/marksman/ ''',
        formatter_class=RawTextHelpFormatter)

    main_parser.add_argument('-l', '--loud',
                             help='increase output verbosity',
                             action='store_true')

    main_parser.add_argument(
        '-v', '--version', action='version', version=__version__)

    subparsers = main_parser.add_subparsers(
        title='actions', help='actions you can take')

    crud_parser = subparsers.add_parser('crud',
                                        help='Do crud operations', )

    email_parser = subparsers.add_parser('email',
                                         help='Email results to students',)

    vis_parser = subparsers.add_parser('visualize',
                                       help='Visualize the results',)

```

```
    utils_parser = subparsers.add_parser(
        'utils', help='Additional utility tools for marksman')

    crud_parser.add_argument('what',
                             help='Choose what data you want to crud',
                             choices=['students', 'exams', 'marks'])
    crud_parser.set_defaults(func=crud_handler)

    email_parser.add_argument('exam',
                              help='exam uid',
                              type=int)
    email_parser.set_defaults(func=email_handler, which='email_parser')

    vis_parser.add_argument('exam',
                            help='exam uid',
                            type=int)

    vis_parser.add_argument('--r',
                            metavar='ROLL',
                            help='roll number of student (default=0 for
all)',
                            type=int,
                            default=0)

    vis_parser.set_defaults(func=visualization_handler, which='vis_parser')

    utils_parser.add_argument('task', help='Choose the task you want to
perform', choices=[
                              'dummy', 'import', 'export'])

    utils_parser.set_defaults(func=utils_handler)

    if len(sys.argv) == 1:
        print(f'marksman {__version__}')
        print('Created by Aahnik Daw.\n')
        main_parser.print_help(sys.stderr)
        sys.exit(0)

    return main_parser.parse_args()


def call_func(args: Namespace) -> None:
    ''' Call the appropriate handler based on args

    Args:
        args (Namespace): the arguments provided at command line
    '''
    if hasattr(args, 'func'):

        logger.info('Starting database connection')
        ensure_parent(DB_PATH)

        pre_exists = os.path.isfile(DB_PATH)

        my_conn = sqlite3.connect(DB_PATH)
        cursor = my_conn.cursor()

        foreign_key_constraint(cursor)
```

```
109
110         if not pre_exists:
111             create_tables(cursor)
112         try:
113             args.func(args, cursor)
114         except Exception as err:
115             logger.exception(err)
116         my_conn.commit()
117         my_conn.close()
118         logger.info('Closed database connection')
119
120
121 def handle_interrupt(*args):
122     ''' Quit Gracefully '''
123     print('\nUser Interrupt recieved. Quitting.')
124     sys.exit(1)
125
126
127 def main():
128     ''' Command line entry point
129     '''
130
131     # handle user interrupt
132     signal.signal(signal.SIGTERM, handle_interrupt)
133     signal.signal(signal.SIGINT, handle_interrupt)
134
135     args = parse_commands()
136
137     if args.loud or LOUD:
138         level = logging.INFO
139     else:
140         level = logging.WARNING
141
142     logging.basicConfig(level=level,
143                         format='[dim]%(name)s[/dim]\t%(message)s',
144                         handlers=[RichHandler(markup=True,
show_path=SHOW_PATH,)])
145
146     logger.info('Verbosity turned on')
147
148     call_func(args)
149
```

db.py

```python
''' A module that implements low level access to the database,
for commonly used SQL operations.'''

import logging
from sqlite3 import Cursor
from marksman.helpers import ___, intify

logger = logging.getLogger(__name__)


def gen_kv_str(kv_dict: dict, delim: str = 'AND') -> str:
    ''' Generate key value strings

    Args:
        kv_dict (dict): dictionary containing keys and values
        delim (str, optional): the stuff to put in between. Defaults to
'AND'.

    Returns:
        str: the rendered key value string
    '''

    key_val_string = ''

    count = 0

    for key, value in kv_dict.items():
        if value:
            if count >= 1:
                key_val_string += f' {delim} '
            if not isinstance(intify(value), int):
                quote = '"'
            else:
                quote = ''
            key_val_string += f'{key} = {quote}{value}{quote}'
            count += 1

    return key_val_string


def create_tables(cursor: Cursor):
    ''' Create all tables required by marksman

    Args:
        cursor (Cursor): a sqlite3 Cursor object
    '''

    logger.info('Starting creation of tables')
    try:
        cursor.executescript(___('''
            DROP TABLE IF EXISTS students ;
            CREATE TABLE students(
                        roll INTEGER NOT NULL PRIMARY KEY,
                        name TEXT NOT NULL,
```

```
                        email TEXT NOT NULL
                                );

            DROP TABLE IF EXISTS exams ;
            CREATE TABLE exams(
                        uid INTEGER NOT NULL PRIMARY KEY,
                        name TEXT NOT NULL
                            );

            DROP TABLE IF EXISTS marks ;
            CREATE TABLE marks(
                        student INTEGER NOT NULL,
                        exam INTEGER NOT NULL,
                        marks INTEGER NOT NULL,

                        FOREIGN KEY (student)
                            REFERENCES students(roll)
                            ON DELETE CASCADE,

                        FOREIGN KEY (exam)
                            REFERENCES exams(uid)
                            ON DELETE CASCADE,

                        PRIMARY KEY (student,exam)
                            );
                ''''))
    except Exception as err:
        logger.warning(
            'Failed to create tables. Delete the database file and try
again')
        logger.exception(err)
    else:
        logger.info('Successfully created tables')


def foreign_key_constraint(cursor: Cursor):
    ''' Turn on foreign key constraints

    Args:
        cursor (Cursor): sqlite3 Cursor object
    '''
    cursor.execute(___('PRAGMA foreign_keys = ON;'))


class DbModelz:
    ''' Class to implement low level access to the database for common
tasks
    '''

    def __init__(self, table: str, cursor: Cursor) -> None:
        ''' Constructor to initialize DbModelz object

        Args:
            table (str): name of the table to work with
            cursor (Cursor): sqlite3 Cursor object
        '''
        self.table = table
        self.cursor = cursor
```

```
        logger.info(f'Created {self}')

    def __str__(self) -> str:
        return f'Modelz object for {self.table}'

    def query(self, query_string: str):
        ''' Execute a query using the Cursor and return all results

        Args:
            query_string (str): the query to execute

        Returns:
            list: results
        '''

        self.cursor.execute(___(query_string))
        return self.cursor.fetchall()

    def fetch(self, **conds) -> list:
        ''' Fetches a list of results based on condition parameters

        Returns:
            list: [description]
        '''

        cond_str = gen_kv_str(conds)
        self.cursor.execute(___(f'''SELECT * FROM {self.table}
                            {'WHERE' if cond_str else ''} {cond_str}'''))
        return self.cursor.fetchall()

    def exists(self, **conds) -> tuple or None:
        ''' Checks whether an entry exists, based on given condition

        Returns:
            tuple or None: returns the tuple containing the entity if
exists
                           else None
        '''

        cond_str = gen_kv_str(conds)
        self.cursor.execute(___(f'''SELECT *
                                FROM {self.table}
                                WHERE {cond_str}'''))

        return self.cursor.fetchone()

    def insert(self, values: tuple) -> None:
        ''' Inserts the values provided in the table

        Args:
            values (tuple): values to insert
        '''

        self.cursor.execute(___(f'''INSERT INTO {self.table}
                                VALUES{values} ''' ))
        logger.info(f'{values} were inserted using {self}')

    def update(self, set_dict: dict, **conds) -> None:
```

```
        ''' Updates the table using set_string, based on given conditions

        Args:
            set_string (str): the string to put beside SET keyword in SQL
        '''

        set_string = gen_kv_str(set_dict, delim=',')
        cond_str = gen_kv_str(conds)
        self.cursor.execute(___(f'''UPDATE  {self.table}
                                SET {set_string}
                                WHERE {cond_str} '''))

    def delete(self, **conds) -> None:
        ''' Deletes the entries satisfying given condtions
        '''

        cond_str = gen_kv_str(conds)
        self.cursor.execute(___(f'''DELETE FROM {self.table}
                                WHERE {cond_str} '''))

```

helpers.py

```python
''' A module that defines various helper functions to avoid repeatition of code
'''

import os
import logging
import sys
from typing import Iterable

from rich.console import Console
from rich.table import Table

from marksman.settings import (DB_PATH, GLOBAL_CONFIG_PATH,
                               SENDER_EMAIL, SENDER_AUTH, SMTP_HOST, SMTP_PORT, INST_NAME)
from marksman.validators import get_email, get_str


logger = logging.getLogger(__name__)

console = Console()


def ___(text: str) -> str:
    ''' Logs a SQL query and returns the query
    To be wrapped around a query string before executing it.

    Args:
        text (str): query string

    Returns:
        str: query string
    '''
    sql_query_logger = logging.getLogger('SQL Executor')
    sql_query_logger.info(f'{text}')
    return text


def not_empty(inp: str) -> str:
    ''' Disallows empty inputs

    Args:
        inp (str): input string

    Returns:
        str: the same input string is returned. If input string is empty, quits the program.
    '''
    if not inp:
        logger.warning('Empty value not allowed. > Quitting ...')
        sys.exit(1)
    return inp


```

```python
def handle_choice(choices: dict) -> None:
    ''' Provides the user with a menu and executes apt function based on user choice

    Args:
        choices (dict): Dictionary containing the option strings as keys
        and the functions to execute when that option is selected as values.

    - this function is specific to my use case and not generic
    - the number of choices in the dict is expected to be one or two and not large
    - all choice str are meant to have unique prefix chars
    - this handler does not resolve prefix collisions, the first come first server
    '''

    ch_list = []
    prefix = '[dim]Choose your action[/dim] '
    msg = prefix
    for char in choices.keys():
        ch_list.append(char[0])
        msg += f'[reverse]{char[0]}[/reverse]{char[1:]} '
    console.print('\n'+msg)
    user_choice = input().lower().strip()
    if not user_choice:
        logger.warning('You did not choose anything')
        return
    for char in choices.keys():
        if char.startswith(user_choice):
            func = choices.get(char)
            func()
            return
    logger.warning('Invalid Choice ...quitting')
    sys.exit(1)


def display_table(title: str, headers: Iterable, table_list: Iterable) -> None:
    ''' Display data in tabular format

    Args:
        title (str): the title of the data
        headers (Iterable): headings or the first row
        table_list (Iterable): the 2D Matrix or list of rows
    '''

    table = Table(title=title, show_lines=True)

    for heading in headers:
        table.add_column(heading, justify='center')

    for row in table_list:
        renderable = [str(c) for c in row]
        table.add_row(*renderable)

    console.print(table)

```

```
105
106 def ensure_parent(filename: str) -> None:
107     ''' Ensures that the parent folder of a file exists
108
109     Args:
110         filename (str): the file name
111     '''
112     parent_folder = os.path.split(filename)[0]
113     os.makedirs(parent_folder, exist_ok=True)
114     logger.info(f'Ensured that parent folder of {DB_PATH} exists')
115
116
117 def save_email_config(thing: str, env_var: str, value: str) -> None:
118     ''' Prompts the use to save a particular configuration related to email
119
120     Args:
121         thing (str): the thing concerned
122         env_var (str): the name of the environment variable dedicated for
    that thing
123         value (str): the value of the thing
124     '''
125
126     text = f'''[red]If you do not want to enter [bold]{thing}[/bold] every
    time,
127     then save it in [blue]{env_var}[/blue] environment variable.[/red]
128         \nYou can write the following line in your {GLOBAL_CONFIG_PATH}
    file:\n
129         [center][dim]{env_var}={value}[/dim][/center]
130         \nRead more about configuring your marksman environment
    http://bit.ly/configure-marksman \n'''
131     logger.warning(text)
132     console.print(
133         f'''Do you want [bold]marksman[/bold] to save your [blue]{thing}
    [/blue] ?
134         [reverse]Y[/reverse]es or [reverse]n[/reverse]o''')
135     choice = input().lower()
136     if choice in 'yes':
137         with open(GLOBAL_CONFIG_PATH, 'a') as _file:
138             _file.write(f'{env_var}={value}\n')
139         console.print(f'Saved [bold]{thing}[/bold] to
    {GLOBAL_CONFIG_PATH}')
140     elif choice in 'no':
141         console.print('Ok. Not saving')
142     else:
143         console.print('Invalid Choice ...not saving')
144
145
146 def configure_email() -> tuple:
147     ''' Help the use configure the settings for emailing
148
149     Returns:
150         tuple: (SENDER_EMAIL, SENDER_AUTH, SMTP_HOST, SMTP_PORT, INST_NAME)
151     '''
152
153     (sender_email, sender_auth,
154      smtp_host, smtp_port,
155      inst_name) = (SENDER_EMAIL, SENDER_AUTH,
156                    SMTP_HOST, intify(SMTP_PORT),
```

```
                    INST_NAME)

    if not sender_email:
        sender_email = not_empty(
            get_email('Enter [bold]sender email[/bold] address: '))
        save_email_config('email', 'marksman_sender', sender_email)
    if not sender_auth:
        sender_auth = not_empty(get_str(
            f'Enter password or auth-code to login into email account
{sender_email}'))
        save_email_config('auth-code', 'marksman_auth', sender_auth)

    if not sender_email.endswith('@gmail.com'):
        if not smtp_host:
            logger.warning(
                '''SMTP sever url could not be derrived from email address.
                Learn more https://git.io/JLMFl ''')
            smtp_host = not_empty(get_str(
                '''Enter address of your email provider\'s SMTP host server
                (keep empty to continue with smtp.gmail.com) : '''))
            save_email_config('smtp host address',
                              'marksman_smptp_host', smtp_host)
        else:
            logger.info(
                f'Sender Email does not end with gmail.com > SMTP_HOST is
{SMTP_HOST}')
    else:
        smtp_host = 'smtp.gmail.com'
        logger.info(f'SMTP_HOST is set to {smtp_host}')

    if not isinstance(smtp_port, int):
        logger.warning(
            'The SMTP Port you have set is not an integer > using default
587')
        smtp_port = 587
    if not smtp_port in (587, 2525):
        logger.warning(
            'The SMTP Port you have set is probably incorrect or insecure')

    return sender_email, sender_auth, smtp_host, smtp_port, inst_name


def intify(string: str) -> int or str:
    ''' Converts a string to int if possible

    Args:
        string (str): input string

    Returns:
        int or str: output int or string
    '''
    try:
        return int(string)
    except ValueError:
        return string


def load_template() -> str:
```

```
    ''' Loads the template file, or provides the default template

    Returns:
        str: the template string
    '''

    templ_file = 'marksman_email_template.md'
    if os.path.isfile(templ_file):
        with open(templ_file, 'r') as file:
            template = file.read()
    else:
        logger.warning(
            'Template file not found. Proceeding with default template')
        template = '''Result and Performance Analysis of ::exam::
        \n### Hi ::name:: you have scored ::marks:: and your rank is
::rank:: in ::exam::.
        \nFind the performance report attached.
        Best regards,
        ::inst::

        _Sent via marksman.mailer_'''
    return template

```

mailer.py

```python
''' Module to implement the email functionality '''

import os
import time
import logging
from smtplib import SMTP
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.mime.base import MIMEBase

from email import encoders
from sqlite3 import Cursor

import markdown
from rich.progress import track

from marksman.plot import analyse_exam, plot_student_performance
from marksman.helpers import ___, load_template


logger = logging.getLogger(__name__)


class Mailer:
    ''' Class to implement related methods for sending emails to students
    '''

    def __init__(self, server: SMTP, cursor: Cursor, sender: str, inst: str, exam_uid: int) -> None:
        ''' Constructor to initialize the mailer object

        Args:
            server (SMTP): SMTP sever object, which is logged in and ready to send mails
            cursor (Cursor): sqlite3 cursor object
            sender (str): email address of sender
            inst (str): institute name of sender
            exam_uid (int): uid of the exam, whose results are to be mailed
        '''
        self.server = server
        self.cursor = cursor
        self.sender = sender
        self.inst = inst if inst else ''
        self.exam_uid = exam_uid
        self.exam_name = self.get_exam_name()
        self.analysis = analyse_exam(self.cursor, self.exam_uid)
        self.template = load_template()
        self.recipient = {}

    def get_exam_name(self) -> str:
        ''' Fetches the name of the exam of the current Mailer object
        Returns:
            [str]: name of exam
        '''
```

```
53
54          self.cursor.execute(
55              ___(f'SELECT name FROM exams WHERE uid={self.exam_uid}'))
56          exam_name = self.cursor.fetchone()[0]
57          return exam_name
58
59      def parse_template(self) -> str:
60          '''
61          Access any variable in your template by ::name:: style
62
63          List of variables passed to template
64              1. name
65              2. roll
66              3. email
67              4. exam
68              5. marks
69              6. rank
70              7. highest
71              8. average
72              9. inst
73
74          Returns:
75              str: the template string
76
77          '''
78          message = self.template
79
80          def merge(dict1, dict2):
81              res = {**dict1, **dict2}
82              return res
83          _vars = merge(self.recipient, self.analysis)
84
85          _vars.update({'exam': self.exam_name, 'inst': self.inst})
86          logger.info(f'varibles supplied to template\n{_vars}')
87
88          for var in _vars.keys():
89              message = message.replace(
90                  f'::{var}::', str(_vars.get(var)))
91
92          return message
93
94      def get_files(self) -> list:
95          ''' Plot the necessary graphs and return a list of file paths
96
97          Returns:
98              list: file paths
99          '''
100
101         path = f'temp/{self.recipient["roll"]}_Performance.png'
102         plot_student_performance(
103             self.cursor, self.recipient['roll'], self.exam_uid,
    self.analysis, path=path)
104         return [path]
105
106     def send_mail(self) -> bool:
107         ''' Send an email to the current recipient
108
109         Returns:
```

```
            bool: status of sending
        '''

        message = self.parse_template()
        recipient = self.recipient['email']
        files = self.get_files()

        multipart_msg = MIMEMultipart("alternative")

        multipart_msg["Subject"] = message.splitlines()[0]
        multipart_msg["From"] = self.inst + f' {self.sender}'
        multipart_msg["To"] = recipient

        text = message
        html = markdown.markdown(text)

        part1 = MIMEText(text, "plain")
        part2 = MIMEText(html, "html")

        multipart_msg.attach(part1)
        multipart_msg.attach(part2)

        for path in files:

            display_file_name = os.path.split(path)[1].split('_')[1]

            time.sleep(2)

            with open(path, 'rb') as _file:
                file_content = _file.read()

            logger.info(f'Attaching {display_file_name}')
            attach_part = MIMEBase('application', 'octet-stream')
            attach_part.set_payload(file_content)
            encoders.encode_base64(attach_part)
            attach_part.add_header('Content-Disposition',
                                   f"attachment; filename=
{display_file_name}")
            multipart_msg.attach(attach_part)

        try:
            logger.info(f'Sending email to {recipient} ...')
            if not recipient.endswith('dontsend@example.com'):
                self.server.sendmail(self.sender,
                                     recipient,
                                     multipart_msg.as_string())
            else:
                logger.warning('Not sending email. Faking Process.')
        except Exception as err:
            logger.warning(f'Failed to send email to {recipient}')
            logger.exception(err)
            return False
        else:
            logger.info(f'Successfully sent email to {recipient}')
            return True

    def mail_all_students(self):
```

```
        ''' Method to loop over all students marks entries for the exam and send them emails.
        Updates the self.recipients and calls self.send_mail in every iteration.
        '''
        self.cursor.execute(
            ___(f'SELECT student,marks FROM marks WHERE exam={self.exam_uid} ORDER BY marks DESC'))
        students_roll_list = self.cursor.fetchall()
        os.makedirs('temp', exist_ok=True)
        rank = 1
        for roll_marks in track(students_roll_list, description='Sending emails'):
            self.cursor.execute(
                ___(f'SELECT * FROM students WHERE roll={roll_marks[0]}'))
            student = self.cursor.fetchone()
            self.recipient['roll'] = student[0]
            self.recipient['name'] = student[1]
            self.recipient['email'] = student[2]
            self.recipient['rank'] = rank
            self.recipient['marks'] = roll_marks[1]

            self.send_mail()

            rank += 1

```

models.py

```python
''' This module implements a generic class to handle the CRUD operations
for 3 different type of data: Students/Exams/Marks
'''

import sys
import logging
from marksman.db import DbModelz
from marksman.helpers import display_table, not_empty

logger = logging.getLogger(__name__)


class Models:
    ''' Generic class for the three types Student or Exam or MarksEntry '''

    def __init__(self, db_modelz: DbModelz, pks_fn: dict, values_fn: dict)
-> None:
        self.db_modelz = db_modelz
        pks = {}
        for k, pk_fn in pks_fn.items():
            pks[k] = pk_fn()
        self.pks = pks  # dict of pk and value

        if not all(self.pks.values()):
            matches = self.db_modelz.fetch(**self.pks)
            self.display(_data=matches)
            sys.exit(0)

        self.values_fn = values_fn  # dict of value_field_name : func
        self.object = self.read()

    def read(self) -> tuple or None:
        ''' Read the data using primary key

        Returns:
            tuple or None: Returns a tuple of the data record, if exists
else None
        '''
        return self.db_modelz.exists(**self.pks)

    def display(self, _data: list = None) -> None:
        ''' Displays the data of the model in a beautified fashion

        Args:
            _data ([list], optional): If _data not provided then
[self.object] will be used.
            Defaults to None.
            This is done to enhance the functionality of this method.
            This method can be used to display a single row or multiple
rows.
        '''
        what = self.db_modelz.table
        if _data is None:
            _data = [self.object]
```

```
        if what == 'students':
            display_table('Student Data', ['Roll', 'Name', 'Email'], _data)
        elif what == 'exams':
            display_table('Exam Data', ['Uid', 'Name'], _data)
        else:  # marks
            display_table('Marks Data', ['Student', 'Uid', 'Marks'], _data)

    def create(self) -> None:
        ''' Create a new entry for model
        '''
        calculated = []
        for _, pk_val in self.pks.items():
            calculated.append(pk_val)
        for _, val_fn in self.values_fn.items():
            val = not_empty(val_fn())
            calculated.append(val)
        self.db_modelz.insert(tuple(calculated))

    def show_related(self) -> None:
        ''' Show data from other tables, that is related to current model
        '''

        what = self.db_modelz.table
        obj = self.object
        if what == 'exams':
            related = self.db_modelz.query(
                f'''SELECT students.roll,students.name,marks.marks
            FROM students
            INNER JOIN marks
            ON students.roll=marks.student
            WHERE marks.exam={obj[0]}
            ORDER BY marks.marks DESC''')
            display_table(f'Students who appeared in {obj[1]}', [
                          'Roll', 'Name', 'Marks'], related)
        elif what == 'students':
            related = self.db_modelz.query(
                f'''SELECT exams.uid,exams.name,marks.marks
                FROM exams
                INNER JOIN marks
                ON exams.uid=marks.exam
                WHERE marks.student={obj[0]}
                ORDER BY exams.uid''')
            display_table(f'Exams given by {obj[1]}', [
                          'Uid', 'Exam Name', 'Marks'], related)
        else:  # marks
            student = self.db_modelz.query(
                f'SELECT name FROM students WHERE roll={obj[0]}')[0][0]
            exam = self.db_modelz.query(
                f'SELECT name FROM exams WHERE uid={obj[1]} ')[0][0]
            display_table('Marks Entry', ['Student', 'Exam', 'Marks'], [
                          (student, exam, obj[2])])

    def update(self) -> None:
        ''' Update data entry of current model
        '''
        calculated = {}
        for k, val_fn in self.values_fn.items():
            calculated[k] = val_fn()
```

```
109         if any(calculated.values()):
110             self.db_modelz.update(calculated, **self.pks)
111         else:
112             logger.warning('You left all values empty > Not updating
    anything')
113 
114     def delete(self) -> None:
115         ''' Delete a data entry of current model
116         '''
117         self.db_modelz.delete(**self.pks)
118 
```

plot.py

```python
''' This module implements plotting of graphs for the purpose of data visualization'''

import logging
from sqlite3 import Cursor
from matplotlib import pyplot as plt

from marksman.helpers import ___


logger = logging.getLogger(__name__)


def analyse_exam(cursor: Cursor, exam_id: int) -> dict:
    ''' Gives all stats about the exam

    Args:
        cursor (Cursor): sqlite3 Cursor object
        exam_id (int): the unique id of the exam

    Returns:
        dict: all the stats
    '''

    logger.info('Analyzing exam with id ')
    cursor.execute(___(f'SELECT AVG(marks) FROM marks WHERE exam={exam_id}'))
    average = cursor.fetchone()[0]
    cursor.execute(___(f'SELECT MAX(marks) FROM marks WHERE exam={exam_id}'))
    highest = cursor.fetchone()[0]

    return {'average': average, 'highest': highest}


def plot_student_performance(cursor: Cursor, roll: int, exam: int, analysis: dict, path='') -> None:
    ''' Plots students performance against the batch

    Args:
        cursor (Cursor): sqlite3 Cursor object
        roll (int): roll no of student
        exam_id (int): unique id of exam
    '''

    cursor.execute(
        ___(f'SELECT marks FROM marks WHERE exam={exam} AND student={roll}'))
    marks = cursor.fetchone()

    if not marks:
        logger.warning(
            f'''Marks entry does not exist ...
            for exam with uid = {exam} and student with roll = {roll} ''')
```

```
        return

    x_axis = ['student', 'highest', 'average']
    y_axis = [marks[0], analysis.get('highest'), analysis.get('average')]

    plt.bar(x_axis, y_axis, width=0.4)

    logger.info(
        f'Plottng horizontal bar grapth with \nx = {x_axis} \n\ny =
{y_axis}')

    for index, value in enumerate(y_axis):
        plt.text(value, index, str(value))

    plt.xlabel('Comparison')
    plt.ylabel('Marks')

    if path:
        plt.savefig(path)
        plt.close()
    else:
        plt.show()


def plot_batch_performance(cursor: Cursor, exam: int) -> None:
    ''' Plot the performance of all students in a particular exam

    Args:
        cursor (Cursor): sqlite3 Cursor object
        exam (int): uid of the exam concerned
    '''
    cursor.execute(
        ___(f'SELECT student,marks FROM marks WHERE exam={exam}'))
    marks_list = cursor.fetchall()

    if not marks_list:
        logger.warning(f'No marks entries exist for exam with uid = {exam}')
        return

    x_axis = [f'{i[0]}' for i in marks_list]
    y_axis = [i[1] for i in marks_list]

    plt.stem(x_axis, y_axis)

    logger.info(f'Plotting stem graph with \nx = {x_axis} \n\ny = {y_axis}')

    plt.xlabel('Students')
    plt.ylabel('Marks')

    plt.show()

```

settings.py

```python
''' Settings and configurations for marksman
'''

import os
from dotenv import load_dotenv

MARKSMAN_DIR = os.path.expanduser('~/.marksman')
GLOBAL_CONFIG_PATH = os.path.join(MARKSMAN_DIR, '.env')
DEFAULT_DB_PATH = os.path.join(MARKSMAN_DIR, 'database.db')

load_dotenv(GLOBAL_CONFIG_PATH)
load_dotenv(override=True)


DB_PATH = os.getenv('marksman_db', DEFAULT_DB_PATH)
SENDER_EMAIL = os.getenv('marksman_sender')
SENDER_AUTH = os.getenv('marksman_auth')
LOUD = bool(os.getenv('marksman_loud'))
SHOW_PATH = bool(os.getenv('marksman_show_path'))
SMTP_HOST = os.getenv('marksman_smtp_host')
SMTP_PORT = os.getenv('marksman_smtp_port', '587')
INST_NAME = os.getenv('marksman_inst')

```

utils.py

```
''' This module implements various extra utilities for marksman.
Such as filling database with dummy data, or importing or exporting CSV.
'''

import os
import logging
import random
import csv
from datetime import datetime

from rich import console
from rich.progress import track
from rich.console import Console

from marksman.validators import get_str
from marksman.db import DbModelz
from marksman.helpers import intify

console = Console()

logger = logging.getLogger(__name__)


def fill_dummy(students: DbModelz, exams: DbModelz, marks: DbModelz) -> None:
    ''' Fill the database with dummy data

    Args:
        students (DbModelz): DbModelz object with table as students
        exams (DbModelz): DbModelz object with table as exams
        marks (DbModelz): DbModelz object with table as marks
    '''

    for i in track(range(1, 51), description='Creating dummy students'):
        students.insert((i, f'stud_{i}', f'{i}dontsend@example.com'))
    logger.info('Created 50 dummy students')

    for j in track(range(1, 11), description='Creating dummy exams'):
        exams.insert((j, f'exam_{j}'))
    logger.info('Created 10 dummy exams')

    for j in track(range(1, 11), description='Creating dummy marks entries'):
        for i in range(1, 51):
            _marks = random.randint(1, 100)
            marks.insert((i, j, _marks))
    logger.info('Filled with random dummy marks entries')

    console.print('Filling dummy data complete!')


class ImportExport:
    ''' A class that implements methods related to import and export of data for marksman.
```

```
    '''

    def __init__(self, students: DbModelz, exams: DbModelz, marks:
DbModelz) -> None:
        ''' Constructor to initialize ImportExport object

        Args:
            students (DbModelz): DbModelz object with table as students
            exams (DbModelz): DbModelz object with table as exams
            marks (DbModelz): DbModelz object with table as marks
        '''
        self.structure = [('students', ('roll', 'name', 'email'),
students),
                          ('exams', ('uid', 'name'), exams),
                          ('marks', ('student', 'exam', 'marks'), marks)]

    def load_csv(self) -> None:
        ''' Read CSV files and insert data into database
        '''
        import_dir = get_str(
            '''Enter the path of the directory which has the csv files:
            ( leave empty if files in current directory )''')
        if not import_dir:
            import_dir = os.getcwd()
        if not os.path.isdir(import_dir):
            logger.warning('Given path is not a directory... quitting')
            return

        for item in self.structure:
            path = f'{import_dir}/{item[0]}.csv'
            try:
                with open(path) as _file:
                    reader = csv.DictReader(_file)
                    for row in reader:
                        values = []
                        for _, value in row.items():
                            values.append(intify(value))
                        item[2].insert(tuple(values))
            except FileNotFoundError:
                logger.warning(
                    f'Could not find {path} > Skipped importing {item[0]}')
            except Exception as err:
                logger.warning(
                    f'''Problem occured while trying to import data from
{path}
                    \nPlease follow specified format.  Read more
http://bit.ly/marksman-utils''')
                logger.exception(err)
            else:
                console.print(f'Finished loading {item[0]} from CSV.')

    def export_csv(self) -> None:
        ''' Read data from database and write them onto CSV files
        '''

        timestamp = str(datetime.now().strftime("%d %b %I:%M %p"))

        export_dir = f'Marksman Export {timestamp}'  # export directory
```

```
        os.makedirs(export_dir, exist_ok=True)

        for item in self.structure:
            with open(f'{export_dir}/{item[0]}.csv', 'w') as _file:
                writer = csv.writer(_file)
                writer.writerow(item[1])
                writer.writerows(item[2].fetch())

        console.print(
            f'Finished exporting to CSV. See the {export_dir} folder.')

    def do_apt(self, task: str) -> None:
        ''' Call the apt method based on provided task argument

        Args:
            task (str): the task
        '''
        if task == 'import':
            self.load_csv()
        else:
            self.export_csv()

```

validators.py

```python
''' This Module has functions validate various user inputs
'''

import sys
import logging
import re

from rich.console import Console

logger = logging.getLogger(__name__)

console = Console()


def get_pos_int(msg: str = 'Enter marks: ') -> int or None:
    ''' Takes input and returns after assuring that it is a positive integer.
        Allows empty input.

    Returns:
        int: the correct integer
    '''
    console.print(f'\n{msg}')
    inp = input()
    if inp:
        try:
            integer = int(inp)
            assert integer > 0
        except ValueError:
            logger.warning('The value you entered is not an integer')
        except AssertionError:
            logger.warning('You must enter an integer greater than zero')
        else:
            return integer
        sys.exit(1)


def get_str(msg: str = 'Enter [bold]name[/bold]: ') -> str or None:
    ''' Takes a string input and assures that it is valid.
        Allows empty input.

    Args:
        msg (str, optional): [description]. Defaults to 'Enter [bold]name[/bold]: '.

    Returns:
        str or None: User input
    '''
    max_len = 64
    while True:
        try:
            console.print(f'\n{msg}')
            string = input()
            assert len(string) <= max_len
```

```
        except AssertionError:
            logger.warning(
                f'The string must not be longer than {max_len} characters')
        else:
            return string


def get_email(msg: str = 'Enter [bold]email[/bold]: ') -> str or None:
    ''' Takes a string from user and assures that it matches the regex of email.
        Allows empty input.

    Args:
        msg (str, optional): [description]. Defaults to 'Enter [bold]email[/bold]: '.

    Returns:
        str or None: User input
    '''
    while True:
        try:
            email = get_str(msg)
            if email:
                logging.info('Checking email for validity')
                regex = r'^[a-z0-9]+[\._]?[a-z0-9]+[@]\w+[.]\w+$'
                assert re.search(regex, email)
        except AssertionError:
            logger.warning('The email you enterd is invalid')
        else:
            return email


def roll() -> int or None:
    ''' Asks the user to enter roll number of student
        .. Note:: This wrapper is essential for certain internal implementation reasons.
    Returns:
        int or None: User input
    '''
    return get_pos_int('Enter [bold]roll number[/bold] of student:')


def uid() -> int or None:
    ''' Asks the user the uid of exam
        .. Note:: This wrapper is essential for certain internal implementation reasons.

    Returns:
        int or None: User input
    '''
    return get_pos_int('Enter the [bold]unique id[/bold] of exam:')

```

Makefile

```
hi:
    @cat Makefile

pypi:
    python3 -m pip install --upgrade pip
    python3 -m pip install setuptools wheel twine
    rm -rf __pycache__
    python3 setup.py sdist bdist_wheel
    twine upload dist/*
    rm -rf build dist *.egg-info

docu:
    pdoc3 --force --html --config show_source_code=False -o ~/temp marksman
```

requirements.txt

```
autopep8==1.5.4
bleach==3.2.1
certifi==2020.12.5
cffi==1.14.4
chardet==4.0.0
colorama==0.4.4
commonmark==0.9.1
cryptography==3.3.1
cycler==0.10.0
docutils==0.16
idna==2.10
jeepney==0.6.0
keyring==21.7.0
kiwisolver==1.3.1
Markdown==3.3.3
matplotlib==3.3.3
numpy==1.19.4
packaging==20.8
Pillow==8.0.1
pkginfo==1.6.1
pycodestyle==2.6.0
pycparser==2.20
Pygments==2.7.3
pyparsing==2.4.7
PyQt5==5.15.2
PyQt5-sip==12.8.1
python-dateutil==2.8.1
python-dotenv==0.15.0
readme-renderer==28.0
requests==2.25.1
requests-toolbelt==0.9.1
rfc3986==1.4.0
rich==9.5.1
SecretStorage==3.3.0
six==1.15.0
toml==0.10.2
tqdm==4.55.0
twine==3.3.0
typing-extensions==3.7.4.3
urllib3==1.26.2
webencodings==0.5.1

```

setup.py

```
import setuptools
import re

with open("README.md", "r") as fh:
    long_description = fh.read()

version = re.search(
    '^__version__\s*=\s*"(.*)"',
    open('marksman/__init__.py', 'r').read(),
    re.M).group(1)


setuptools.setup(
    name="marksman",
    version=version,
    author="Aahnik Daw",
    author_email="meet.aahnik@gmail.com",
    description="CLI Tool to manage marks of students efficiently.",
    long_description=long_description,
    long_description_content_type="text/markdown",
    url="https://github.com/aahnik/cbse-xii-cs-proj/tree/main/project",
    packages=setuptools.find_packages(),
    install_requires=['matplotlib',
                      'PyQt5',
                      'Markdown',
                      'rich',
                      'python-dotenv'],
    include_package_data=True,
    zip_safe=False,
    entry_points={
        'console_scripts': ['marksman=marksman.cli:main',
'mm=marksman.cli:main'],
    },
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
    ],
    python_requires='>=3.8',
    license='MIT License',
    keywords=['sql', 'database', 'crud', 'marks management',
              'cms', 'data management', 'cli', 'utility', 'tool'],
)

```

Bibliography

Books:

1. Computer Science with Python (Class XII) by *Sumita Arora*

Web Resources:

The links are in no particular order.

1. https://matplotlib.org/
2. https://rich.readthedocs.io/en/stable/introduction.html
3. https://realpython.com/python-send-email/
4. https://github.com/theskumar/python-dotenv
5. https://docs.python.org/3/howto/argparse.html
6. https://sqlite.org/index.html
7. https://docs.python.org/3/library/sqlite3.html
8. https://pylint.org/
9. https://python-packaging.readthedocs.io/en/latest/
10. https://packaging.python.org/tutorials/packaging-projects/
11. https://docs.python.org/3/library/argparse.html#sub-commands
12. https://www.geeksforgeeks.org/python-os-getenv-method/
13. https://docs.python.org/3/library/logging.html

The sub-pages of the above web pages are also to be included, when not mentioned explicitly.