

Data Parser report

The structure of the project is outlined in Figure 1 below.

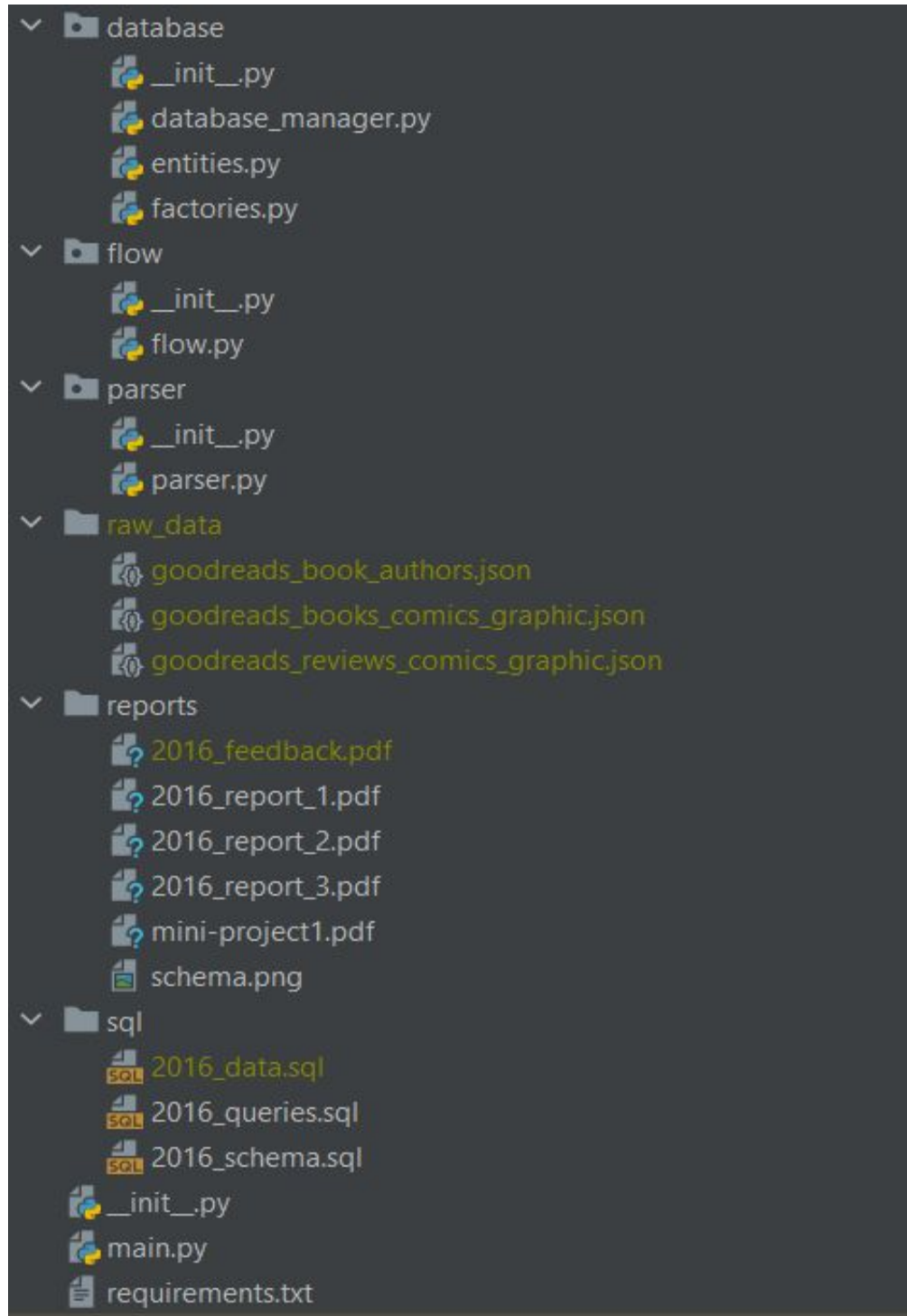


Figure 1: Project structure

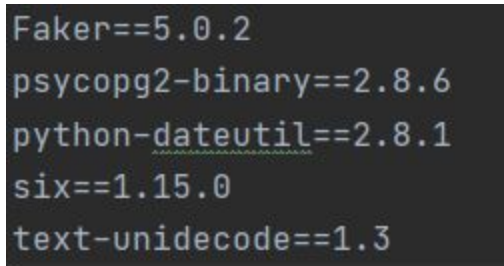
Design practises and overview:

Prerequisites:

The project was written in **Python 3.8.2**, it uses some new features of python 3.8 such as the [walrus operator](#), so it will work only with python **3.8** and above.

The package requirements can be found in the *requirements.txt* file shown in Figure 2. They can be installed to your python environment via the command:

pip install -r requirements.txt



```
Faker==5.0.2
psycpg2-binary==2.8.6
python-dateutil==2.8.1
six==1.15.0
text-unidecode==1.3
```

Figure 2: Contents of requirements.txt

A postgres database, should be already created with an **active schema** and **no data**. Details on how to create the schema can be found in *2016_report_1.pdf* file

the raw_data directory must contain the json data files as shown in the Figure 1:

goodreads_book_authors.json
goodreads_books_comics_graphic.json
goodreads_reviews_comics_graphic.json

Program assumptions + algorithms

The code is commented upon sufficiently so I will not delve into many details.

The main assumption on which the whole script is based, is that the data included in the following files:

goodreads_book_authors.json
goodreads_books_comics_graphic.json
goodreads_reviews_comics_graphic.json

is real. So no fake data is created for any missing data by the parser initially.

Directories:

parser:

Contains the `parser.py` file which implements the parsing logic, done by *UCSDJsonDataParser* class.

The algorithm that is used for parsing is as follows:

Executed by the method:

process_data

1. Parse *goodreads_book_authors.json* line by line
2. Process authors and store only the valid ones in memory
3. Parse *goodreads_books_comics_graphic.json* line by line
4. Process books, for each one that is valid
5. Store its publisher (if valid), its (authors if valid), associate it with its authors (if any valid are found)
6. Store the processed book and its processed relations in memory
7. Parse *goodreads_books_comics_graphic.json* line by line
8. Process reviews and store only the valid ones in memory
9. Associate each valid review with the book it refers to

database:

Contains the *database_manager.py*, *entities.py* and *factories.py* files.

database_manager.py: Contains the *ComicBooksDBManager* class which handles all the connections to the database and also inserts the parsed data into the database.

The algorithm used for data insertion in the **main flow** is as follows:

Executed by the method:

insert_parsed_data

1. Insert into the database all the authors that are parsed
2. Keep a mapping between their parsed ids and the ones that are used in the database
3. Insert each book and its relations.
 - a. Insert each publisher (if available)
 - b. Insert each book
 - c. Since authors are already inserted, use the mapping from step 2 in order to get their database ids and insert the data into the `book_author` table accordingly. (if available)
 - d. Insert the reviews into the review table and insert into `book_review` table accordingly (if available)

The algorithm used for data insertion in the **test flow** is as follows:

Executed by the method:

`create_test_data(self, user_num=10, order_per_user=5, address_per_user=3)`

1. Update the book current_price table `user_num` x `order_per_user` times with a randomly generated price, and print the number of books picked. Notice that the number printed represents the book ids that have been updated. For example if it is 50, it means that the book ids from 1 to 50 have their current_price field altered.
2. Create `user_num` users with fake data and insert them into the database
3. Create `address_per_user` x `user_num` addresses and insert them into the database
4. Associate each user with `address_per_user` addresses and insert them into the database
5. Create `user_num` * `order_per_user` orders and insert them into the database
6. Create `user_num` * `order_per_user` book orders and insert them into the database

entities.py: All database entities are represented as objects in this file, so that the data can be manipulated more efficiently. Figure 3, illustrates an example of the User entity.

```
class User(BaseEntity):
    """Represents a User entity"""

    def __init__(self):
        self.username = None
        self.password = None
        self.phone_number = None
        self.email = None
        self.real_name = None

    def __str__(self):
        return f"User(username={self.username})"

    def to_dict(self):
        return {"username": self.username, "password": self.password, "phone_number": self.phone_number,
                "email": self.email, "real_name": self.real_name}
```

Figure 3: User object in entities.py

factories.py: Utilizes the *faker* package in order to create entity objects with fake data. *FakeGenerator* class is used in order to generate the data. Figure 4, illustrates an example of the UserFactory.

```
class UserFactory(object):
    """Class used for generating fake User data"""

    @staticmethod
    def generate_users(n=1):
        """
        Generator of User objects
        :param n: number of objects to be generated
        """
        emails = list(_fg.emails(n=n))
        usernames = list(_fg.usernames(n=n))
        for i in range(n):
            data = {"username": usernames[i], "email": emails[i], "password": _fg.password(),
                    "real_name": _fg.name(), "phone_number": _fg.phone_number()}
            yield User.build_from_data(data)
        _fg.clear_unique()

    def __str__(self):
        return "UserFactory"
```

Figure 4: UserFactory object in factory.py

The program is splitted into 3 flows, located in *flow.py*. Each flow can be pictured as a separate script. They are as follows:

1. **main**: parses the dataset and inserts the actual data into the db
2. **test**: provided that the main flow has been executed at least once or the database contains the data from the main flow, creates some users, orders and addresses for testing.
3. **test_rb**: clears the tables that were used by the test in order for the database to contain clean data, however book price updates have to be cleaned manually.

raw_data:

contains the json files from the dataset.

reports:

contains the report pdf deliverable files along with a photo of the schema

sql:

contains the sql deliverable files.

Notice: *2016_data.sql* is not created by the parser, but can be easily generated after executing the main script and then using *pg_dump* to export the data

Example command:

```
pg_dump -d comic_books -a --no-owner -f 2016_data.sql
```

Execution instructions

The execution script is named ***main.py*** and calls only one function:

run_exercise()

Figure 3, illustrates the execution parameters required by the program.

Each one is described by calling the *--help* option.

```
17:39 $ python main.py --help
usage: main.py [-h] -d DATABASE -pwd PASSWORD [-u [USER]] [-i [IP]] [-p [PORT]] [-f {main,test,test_rb}]

optional arguments:
  -h, --help            show this help message and exit
  -d DATABASE, --database DATABASE
                        the name of the database
  -pwd PASSWORD, --password PASSWORD
                        password for the specified database user
  -u [USER], --user [USER]
                        database user, defaults to postgres
  -i [IP], --ip [IP]    connection ip, defaults to localhost
  -p [PORT], --port [PORT]
                        connection port, defaults to 5432
  -f {main,test,test_rb}, --flow {main,test,test_rb}
                        flow of the program, defaults to main. It has 3 options: main: parses the dataset and inserts
                        the actual data into the db, test: provided that the main flow has been executed at least once
                        or the database contains data creates some users, orders and addresses for testing, test_rb:
                        clears the tables that were used by the test in order for the database to contain clean data
```

Figure 3: Execution of main.py

Execution examples:

1. *python main.py -d comic_books -pwd root*
2. *python main.py -d comic_books -pwd root -f main*
3. *python main.py -d comic_books -pwd root -f test*
4. *python main.py -d comic_books -pwd root -f test_rb*
5. *python main.py -d comic_books -u foo -ip 0.0.0.0 -p 1234 -pwd foobar*