API Code Documentation

The entire API codebase is divided into files.

FILES

There are 4 files that are of high importance

- App.py
    - Contains all the web application code for the Flask Application and REST-API

- Data_scraper.py
    - Contains the selenium code used to scrape data for the dataset

- Database_access.py
    - Contains the python and SQL queries to initialize the SQLite3 Database, and keep updating it, with updations of data.

- Hash_map_operations.py
    - Contains all the python code responsible for creating, updating, accessing and read the hash-table data structure.

- String_operations.py
    - Uses NLP techniques to generate the synsets and is responsible for parsing the strings.

METHODS

Within **App.py**

There are 3 methods

```
home():
```

- Acts as the landing page for the Web Application and grants access to update the SQLite3 Database.

```
update_db()
```

- This method is called whenever the submit button is pressed on the render template.
- Updates the database with user given values.
- Contains a trigger for the updation of the hash table whenever inputs are submitted.
- Trigger implemented through the function `dynamic_data_dump(par1,par2)`
  - `par1` is the product_number / product_id
  - `par2` is the product_desription

```
general_search()
```

- This method calls the REST-API from the website link, and json is returned.

Within **data_scraper.py**

There are 2 methods

```
show_me_the_money(par1)
```

- Takes `par1` as a list of websites to scrape.
- Prints out the CSV form all the data that was scraped.
- This can be directly fed into a .csv file.
- I named the .csv file as usable_data.csv

```
file_len(par1)
```

- Takes `par1` as the name of the file, to check the number of lines of the file.
- Returns back the length of the file.

Within **database_access.py**

There are 1 method

```
initial_data_dump_in_db()
```

- Reads data from usable_data.csv and writes line by line into the database.

Within **hash_map_operations.py**

There are 6 methods

```
initial_data_dump()
```

- Dumps all the values of data into the hash-table

- Makes use of functions `update_hash_map(par1,par2)` and `parse_string(par3)` where `par1` is a list of all identified synnet words, `par2` is product_id, and `par3` is the parsed product description.

```
dynamic_data_dump(par1, par2)
```

- `par1`, and `par2` are product_id and customer query string respectively.
- Makes use of functions `update_hash_map(par1,par2)` and `parse_string(par3)` where `par1` is a list of all identified synnet words, `par2` is product_id, and `par3` is the new product's description.

```
create_hash_map()
```

- Enables creation of a hash-table and inserts one dummy value that won't affect it in an adverse way..

```
update_hash_map(par1,par2)
```

- `par1` is a list of all identified synnet words and `par2` is product_id.
- This is used to update the key value pairs in the hash-table

```
read_hash_map()
```

- `par1` is a list of all identified synnet words and `par2` is product_id.
- Enables the printing out of all the key value pairs in the hash-table.
- Not recommended for use if hash-table is not small.

```
access_hash_map(query)
```

- `query` is a customer query that contains parsed and formatted words.
- Extensively used in the API method `general_search()`

Within **string_operations.py**

There are 2 methods

```
all_syno(par1)
```

- Takes `par1` as a word.
- Returns a list of all words in that corresponding synnet.

```
parse_string(par1)
```

- Takes `par1` as the product description as a string.
- Returns back the processed list of the string.