December 19, 2024

CBS COPENHAGEN BUSINESS SCHOOL
HANDELSHØJSKOLEN

Bok-Bok: Design and Simulation of a Book Exchange Platform

Report

Programming, Algorithms and Data Structures

Program: MSc in Business Administration and Data Science

Author: Michele Daconto (176193)

Teacher: Somnath Mazumdar

Number of characters: 18,307

Number of pages: 12

Academic Year 2024/2025

Table of Contents

Abstract

The principal aim of this report is to provide the reader with a comprehensive guide to the code presented in the notebook 'Bok-Bok_final_assignment_176193.ipynb,' elucidating the rationale behind each architectural decision made during its development. The program is an autonomous simulation of an online book exchange platform, aptly named 'Bok-Bok', designed by the author to demonstrate his solid and in-depth understanding of the Python programming language, acquired throughout the course.

The project exemplifies the practical applications of various theoretical concepts covered in class, including object-oriented programming (OOP) and error handling using the try-except construct, as well as an integration with an API. In particular, the program's linkage to the Google Books API permits the processing of authentic and updated data, thereby markedly enhancing its functionality. With minor modifications, in fact, the program could be transformed into an interactive platform and could serve as a foundation for a complete application.

Keywords: Google Books API - Object-Oriented Programming (OOP) – Exchange Scenarios – Error Handling – Simulation Metrics.

1. The Idea

The underlying idea of this project is to develop a program capable of autonomously simulating the basic operation of a book-swapping application. Who's wishing to initiate the simulation only needs to specify the number of users to be generated, the number of books and the number of exchanges. Once these inputs are provided, the code will produce an output detailing all successful exchanges as well as those rejected by one of the users. At the end of the simulation, if more than ten exchanges are involved, the program will also provide summary metrics describing the results of the simulation.

To enable the code to work with real data about books, it was integrated with an API.

1.1 Google Books API

An API (Application Programming Interface) is an interface that enables different software systems to communicate with each other to exchange data. A practical example is provided by the Google Maps APIs: many applications, such as delivery or sharing services, use these APIs to integrate functionalities such as geolocation, route calculation, or map visualization directly into their platforms. This approach allows them to avoid developing a mapping system from scratch.

In the project, I utilized an API also developed by Google, which does not provide map-related data but rather information about books. The Google Books API is arguably one of the most popular in the field of books and is continuously updated with new releases. It is also free up to the "free quota" of 1,000 requests per day. This is why who is running the simulation must be careful not to exceed this limit. It is also for this reason that I have set the maximum number of books that can be generated in a single simulation to 960, ensuring that all requests are not exhausted in a single run.

Our program communicates with the API in this way: it sends HTTP requests (using the *requests* library imported at the beginning of the code) to the API, which responds with data in JSON format if the user-provided key is authenticated and valid. However, the Google Books API can only handle up to 40 requests at a time. For this reason, when the simulation involves more than 40 books, a *for* loop was implemented to send multiple requests. For example, if 210 books need to be generated then six requests must be sent, as five requests would only return data for 200 books.

What information does the Google Books API provide? Look at the documentation to understand this better. There is a wide range of data available, but certain pieces of information have not been included in order to avoid making the project too complex.

2. The Object-Oriented Approach

Object-oriented programming (OOP) is a programming paradigm that employs the use of objects to organize code. Objects are instances of classes, and they encapsulate both data (attributes) and behaviors (methods) thereby promoting the principles of modularity, reusability and scalability.

In the development of Bok-Bok, OOP mechanisms were employed to model the key entities: users, books, and exchanges. Each of these entities is a class in the code and represents a concrete concept which includes attributes for storing data (such as the title of a book, the ID of a user, details about an exchange…) as well as methods for carrying out actions (such as performing an exchange or extracting books from the API). The utilization of this kind of approach ensures that the platform's codebase is both clean and organized.
Let’s examine the purpose and functionality of each class together.

2.1 Book Class

The Book class has been designed to encapsulate the data for each book.
Using the *get_classics()* method, which includes error handling via a *try-except* construct for cases where the simulation involves more than 40 books or less, the program connects to the Google Books API and populates each instance of Book (i.e. each book) with the following attributes:

- title
- author(s)
- publisher
- description
- published_date
- pages
- isbn (unique identifier for each title)

Analyzing the *get_classics()* method, notice that the requests sent to the API always specify the parameter search_term = "classics". This is because the API allows us to filter requests, and I deemed it appropriate to use this functionality to run the simulation using books that are as well-known as possible - specifically, great literary classics.

Once all its attributes have been updated, each book is added to the *offered_books[]* list, which consequently contains all the books offered for exchange by users on the platform. If the API does not provide data for a particular attribute of a given book, the program will return "Unknown" as the default value.

2.2 User Class

The User class is designed to store information about each user of the platform. Of the three main classes, it is the simplest, containing only a few easy attributes and a straightforward method.
Each instance of a user will have the following attributes:

- full name: randomly generated using the *names* library imported at the top of the code
- email: the user's email address
- owned_books: a list of books owned by the user (max. 30)
- wishlist: a list of wished books (max. 50)
- user_id: a unique identifier for each user, starting with user_id = 1 for the first user and increasing sequentially
- exchanges_completed: a counter that tracks the number of exchanges completed by the user

The *owned_books* and *wishlist* attributes are randomly generated based on the *offered_books[]* list from the Book class. However, the randomization process is not completely arbitrary or unregulated. Certain scenarios are considered, such as when the number of *offered_books* is less than 30, to ensure that no errors occur when assigning books to *owned_books*. Similarly, if the number of *wishlist_candidates* - books available on the platform but not yet owned by the user - is less than 50, this is also managed to ensure that the wishlist is created correctly. These measures ensure that the logic remains consistent, and the user profiles are realistic.

The sole method in the class is *will_exchange()*, a straightforward function that returns *True* with a probability of 66.67% and *False* with a probability of 33.33%. This method is used to determine whether a user is willing to exchange books (True) or not (False).

Each user, along with their attributes, is then added to the *all_users[]* list, which contains all the users generated on the platform.

2.3 Exchange Class

The Exchange class is undoubtedly the most complex, as it fully manages each exchange, whether it is successfully completed or not.
Each instance of an exchange contains the following attributes:

- user1: a randomly selected user from the *all_users[]* list in the User class.
- user2: another randomly selected user from the *all_users[]* list, different from user1, since an exchange must involve two different users.
- book1: the book that user1 gives to user2, initially set to *None*, since its assignment depends on the scenario between the two users.
- book2: similarly to book1, this is the book that user2 gives to user1, also initially set to *None*.
- balancing: a monetary adjustment ranging from a min. of €2 to a max. of €10, included in the exchange with a probability of 33.33%.
- payment_method: a randomly chosen payment method from a selection of popular options (e.g. PayPal). Set to None when balancing = 0.
- context: a string describing the scenario defined by the two users, based on their *owned_books* and *wishlists*.

This class is designed to handle the complex interactions and conditions that arise during exchanges between users. The *perform_exchange()* method manages the exchange process in a structured and detailed way.
First, the method checks whether both users are currently willing to exchange books using the *will_exchange()* function. If at least one of the two users is not willing, the program records the exchange as declined. If both users are willing to exchange, the method evaluates four possible scenarios based on the intersections of *owned_books* and *wishlist* for both users:

- User1's wishlist contains books owned by User2, and User2's wishlist contains books owned by User1.

  In this case:

- Book1 is randomly selected from the books owned by user1 that are in user2's wishlist.
- Book2 is randomly selected from the books owned by user2 that are in user1's wishlist.

- User1 owns a book that is in User2's wishlist, but User2 does not own a book in User1's wishlist.

In this case:

- Book1 is chosen at random from the books owned by User1 that are in User2's wishlist.
- Book2 is chosen at random from the books owned by User2.

- User2 owns a book that is in User1's wishlist, but User1 does not own a book in User2's wishlist.

In this case:

- Book1 is chosen at random from the books owned by User1.
- Book2 is randomly selected from the books owned by User2 that are in User1's wishlist.

- Neither User1's wishlist contains any book owned by User2, nor does User2's wishlist contain any book owned by User1.

In this scenario the exchange will still take place:

- Book1 is randomly selected from the books owned by User1.
- Book2 is randomly selected from the books owned by User2.

These scenarios, described in the context attribute, are mutually exclusive, i.e. if one scenario applies, none of the others can occur at the same time.

After the books (book1 and book2) have been assigned, the exchange is completed by updating the users' book collections as follows:

- Book1 is added to user2's *owned_books* and removed from their *wishlist*, if any.
- Similarly, book2 is added to user1's *owned_books* list and removed from their *wishlist*, if any.

This ensures that both users' book collections and wishlists are updated consistently after the exchange. The *perform_exchange* method either completes the exchange successfully or logs it as failed.

The *display_exchange()* method is responsible for displaying the details of an exchange in the console for those running the simulation. The method provides two different types of output: one for exchanges that were completed without a monetary adjustment, and another for exchanges that involved a monetary transaction between the two users.

Although the two outputs are similar, they are tailored to reflect whether or not a monetary adjustment took place. Both aim to clearly describe the users and books involved in the exchange, providing concise information about the books such as title, author(s), publisher, published date and ISBN. This makes the output both informative and easy to understand.

```
     ------------------------------ Exchange #10 ------------------------------
David Peterson (ID: #5) received 'Manga Classics: The Stories of Edgar Allan Poe' from Melanie Bremmer (ID: #4)
Melanie Bremmer (ID: #4) received 'Celebrate Literature Reader 2' from David Peterson (ID: #5)
No money involved in the transaction.
The books were not on each other's wishlists.

INFO: 'Celebrate Literature Reader 2':
Author(s): Unknown, Publisher: Pearson Education India, Date: Unknown, ISBN: 8131755584.

INFO: 'Manga Classics: The Stories of Edgar Allan Poe':
Author(s): Edgar Allan Poe, Stacy King, Publisher: Manga Classics, Date: Unknown, ISBN: Unknown.
     ------------------------------------------------------------------------------------------
```

Output of an exchange without a money transaction involved

```
     ------------------------------ Exchange #2 -------------------------------
Lila Coleman (ID: #10) received 'Tutti i romanzi' from Elva Smith (ID: #8)
Elva Smith (ID: #8) received 'The Belly Fat Cure Sugar & Carb Counter REVISED' from Lila Coleman (ID: #10)
Money (in €) involved in the transaction: 9.56 from Elva Smith to Lila Coleman (Payment Method: PayPal)
Only 'The Belly Fat Cure Sugar & Carb Counter REVISED' was on Elva Smith's wishlist.

INFO: 'The Belly Fat Cure Sugar & Carb Counter REVISED':
Author(s): Jorge Cruise, Publisher: Hay House, Inc, Date: 2012-10-15, ISBN: 1401940811.

INFO: 'Tutti i romanzi':
Author(s): Jane Austen, Publisher: Newton Compton Editori, Date: 2010-11-05, ISBN: 8854125768.
     ------------------------------------------------------------------------------------------
```

Output of an exchange with a money transaction involved

3. The Simulation

Having defined how the code should handle each object within the classes, we can proceed to build the *simulation()* function. This function orchestrates the overall operation of the Bok-Bok platform by optimally combining all the components created within the classes.
The *simulation()* function takes three input parameters:

1. The number of users to be generated
2. The number of books to be generated (not more than 960)
3. The number of exchanges the user wishes to simulate

As a first step, the *simulation()* function uses the *get_classics()* method to retrieve book data from the API. If, for some reason, *get_classics()* fails to retrieve any books - for example, the user requests a simulation with 0 books - the program will stop and exit the function. In such cases, a *try-except* construct ensures that the error is handled gracefully by printing the message: "Error: the API couldn't fetch any book".

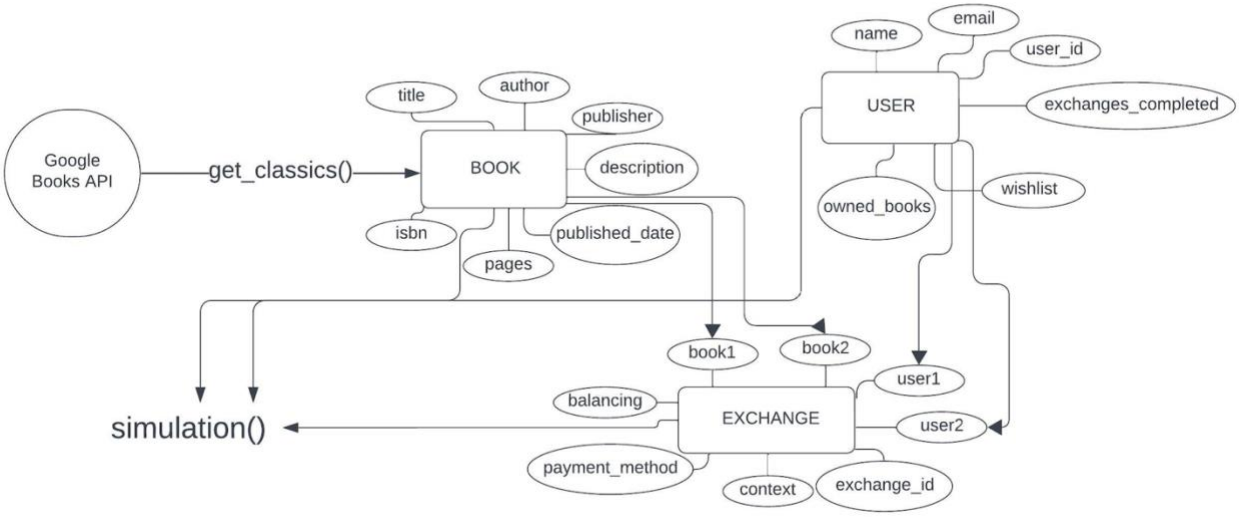
A similar approach is used for user generation. If the simulation involves fewer than two users, or fewer than two books, it will be obvious that no exchanges can take place: a single user cannot swap books alone, and two users cannot swap the only book available on the platform. In such cases, the simulation stops, again using a *try-except* construct, and prints: "Error: not enough books or users to allocate owned_books and wishlist, please generate more books or users".
These logics ensures that the simulation only continues under conditions where exchanges are feasible and meaningful.

The *simulation()* function includes a similar validation check for exchanges. If the user attempts to run a simulation with 0 exchanges, the program will print: "Error: the simulation can't run without exchanges".

Before starting the simulation, the function performs a final check on the maximum number of books that can be generated. If the user specifies a simulation with more than 960 books, the simulation will continue with a 960-books simulation. This is because we want to prevent the person running the simulation from depleting all requests in a single attempt. In this case, the following warning will be printed: "WARNING: you cannot generate that number of books. Proceeding the simulation with 960 books".

After these checks, exchanges are created, and each exchange is executed and displayed in the console using the *perform_exchange()* and *display_exchange()* methods.

To start the simulation, the user is prompted to enter the number of users, books and exchanges he/she wishes to generate. The simulation then runs autonomously. Input validation is handled by a *try-except* construct to ensure that the input is numeric. If the user enters strings or other incompatible data types, the input is rejected with an appropriate error message, ensuring a smooth and error-free execution process.



Architecture of the programme

3.1 Metrics

If more than 10 exchanges are generated, metrics are displayed at the end of the simulation to summarize what has happened. These metrics provide insight into the results of the simulation and user interactions.

The metrics included are:

- Percentage (%) of users not involved in an exchange
- Number of refused exchanges
- Average number of books owned by each user
- Average number of books on each user's wishlist
- Percentage (%) of exchanges involving two books, both on the user's wishlist

- Percentage (%) of exchanges involving only one book in a user's wishlist
- Percentage (%) of exchanges involving books that are not on either user's wishlist
- Percentage (%) of exchanges that resulted in a monetary adjustment
- Percentage (%) of monetary adjustments by payment method
- Percentage (%) completed using credit/debit card
- Percentage (%) completed using PayPal
- Percentage (%) completed using Apple Pay
- Percentage (%) completed using Google Pay
- Percentage (%) completed using Samsung Pay
- Percentage (%) completed using cash

These metrics provide a comprehensive overview of user behaviors, book exchanges and payment trends within the simulation, helping the reader to analyze and understand the dynamics of the platform.

3.2 Possible Future Improvements

As mentioned at the beginning, the simulation created can be used as a basis for implementing a more complex application. In general, the *random* commands should be replaced with *input* commands requested directly to the users (e.g. each user manually enters the books they own and want, rather than having them randomly assigned as in the simulation). In addition, more advanced features could be implemented, such as a search bar that allows users to find other users or specific books. It would also be interesting to allow users to create thematic communities (e.g. around the Harry Potter series) where they can discuss and share books with other enthusiasts.

4. Achieved Learning Objectives

Completing this project was an invaluable opportunity for me, as it allowed me to test my skills in the Python programming language by tackling the management and implementation of a relatively long and complex code base. I learned how to connect an API to a program, a task I had never attempted before but which is now an essential asset as it allows me to craft much realistic programs closely integrated with real world data. I made effective use of the *names*, *requests* and *random* external libraries as they were crucial for the developing of the code.

Further, I consolidated my object-oriented programming capabilities, previously honed during the scooter-sharing platform assignment, by constructing classes capable of efficiently and robustly containing data while avoiding redundant code. Within the *Exchange.select_books()* method, I developed a functional book selection algorithm that allows participants in an exchange to trade books, prioritizing those most desired by each user. In addition, the algorithm can detect scenarios in which there is no exchange due to rejection of the proposal by one of the participants.

The delivered code is the result of many hours of work, and solving the challenges it presented was not always straightforward. In particular, I spent a great deal of time successfully connecting the API, especially when extracting data from nested dictionaries within the JSON response, such as retrieving the ISBN for each book.

Finally, this project improved both my technical and problem-solving skills in software development and represented an important step in my learning journey.

References

- Python Software Foundation, *Python 3.12 documentation*, Python Software Foundation, 2024.
  https://docs.python.org/3.12/
- Google, *Using the Books API*, Google Developers, (n.d.).
  https://developers.google.com/books/docs/v1/using
- Real Python, *The GET request*, Real Python, (n.d.).
  https://realpython.com/python-requests/#the-get-request
- Python Software Foundation, *random – Generate pseudo-random numbers*, Python Documentation, (n.d.).
  https://docs.python.org/3/library/random.html
- Klopper T., *names: a Python library for generating random names*, GitHub, (n.d.).
  https://github.com/trevorstephens/names
- Lott S., *Mastering object-oriented Programming (2nd edition)*, Packt Publishing, 2019.