ChapterOne

Group 5, “CodeFirstGirlsAloud”:
Lucy Tesco, Jessica Wan, Millie Davidson, Jesse Musenge, Olivia Boddy, Elena Newell

Introduction:

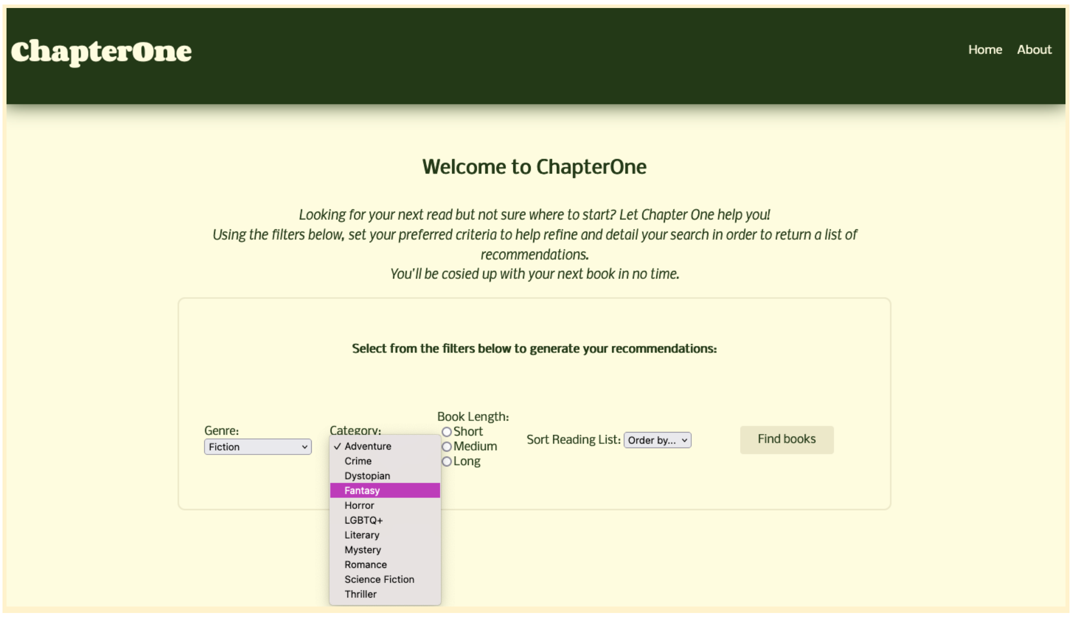
Aims and objectives of the project
We wanted to build a website that would allow users to input search criteria and receive back a list of recommended reading. For example, a user could input a genre, book length and publication time frame and receive back a tailored list of books.

In building this project our key objectives were to:

- Create an easy to use and accessible front-end website that accepted user input
- Take the user input to create customised endpoints in Python that searched a relevant API
- Call the API to retrieve data of books that matched the user criteria
- Return the resulting book list to the front-end in a presentable manner for the user

Background:

In order to try and alleviate the daunting task of where to start looking for book recommendations, we decided to create a website that generates a recommended reading list of ten books based on a selected genre and additional filters as desired by the user. Our intended audience is adult readers with a desire to discover and enjoy a new book. The purpose of the website is to provide a small, focused recommended reading list based on a specific user-selected genre, to give users helpful, informed and detailed suggestions on what to read.



Welcome to ChapterOne

Looking for your next read but not sure where to start? Let Chapter One help you!
Using the filters below, set your preferred criteria to help refine and detail your search in order to return a list of recommendations.
You'll be cosied up with your next book in no time.

Select from the filters below to generate your recommendations:

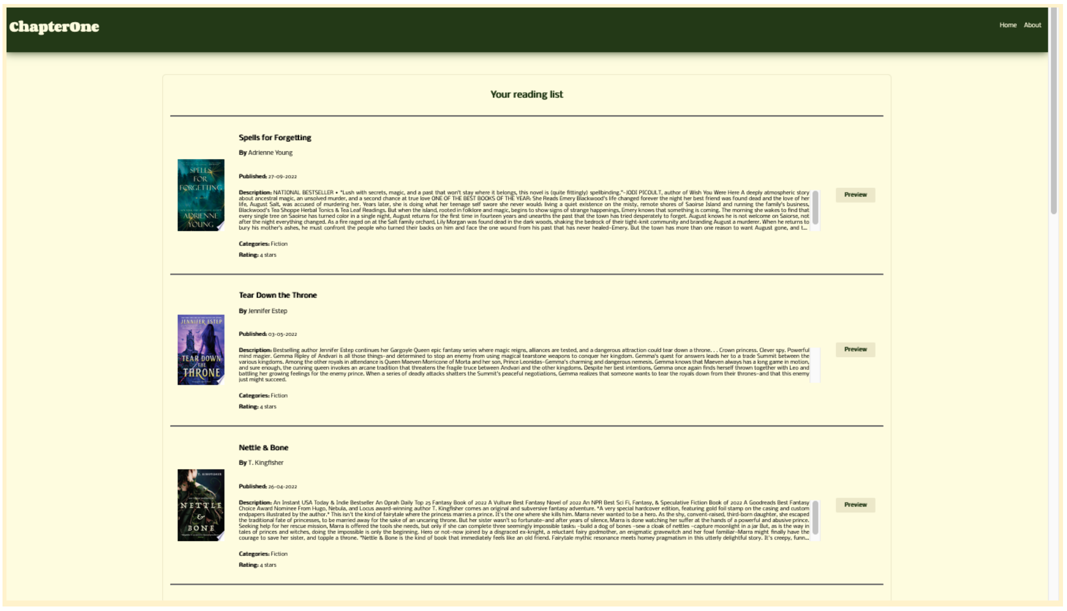
Genre: Fiction

Category: ✓ Adventure, Crime, Dystopian, Fantasy, Horror, LGBTQ+, Literary, Mystery, Romance, Science Fiction, Thriller

Book Length: Short, Medium, Long

Sort Reading List: Order by...

The homepage of ChapterOne website, showing the filter options


ChapterOne

Home About

Your reading list

Spells for Forgetting

By Adrienne Young

Published: 27-09-2022

Description: NATIONAL BESTSELLER • "Lush with secrets, magic, and a past that won't stay where it belongs, this novel is (quite fittingly) spellbinding."–JODI PICOULT, author of Wish You Were Here A deeply atmospheric story about ancestral magic, an unsolved murder, and a second chance at true love ONE OF THE BEST BOOKS OF THE YEAR: She Reads Emery Blackwood's life changed forever the night her best friend was found dead and the love of her life, August Salt, was accused of murdering her. Years later, she is doing what her teenage self swore she never would: living a quiet existence on the misty, remote shores of Saoirse Island and running the family's business, Blackwood's Tea Shoppe Herbal Tonics & Tea Leaf Readings. But when the island, rooted in folklore and magic, begins to show signs of strange happenings, Emery knows that something is coming. The morning she wakes to find that every single tree on Saoirse has turned color in a single night, August returns for the first time in fourteen years and unearths the past that the town has tried desperately to forget. August knows he is not welcome on Saiorse, not after the night everything changed. As a fire raged on at the Salt family orchard, Lily Morgan was found dead in the dark woods, shaking the bedrock of their tight-knit community and branding August a murderer. When he returns to bury his mother's ashes, he must confront the people who turned their backs on him and face the one wound from his past that has never healed–Emery. But the town has more than one reason to want August gone, and t...

Categories: Fiction

Rating: 4 stars

Preview

Tear Down the Throne

By Jennifer Estep

Published: 03-05-2022

Description: Bestselling author Jennifer Estep continues her Gargoyle Queen epic fantasy series where magic reigns, alliances are tested, and a dangerous attraction could tear down a throne. . . Crown princess. Clever spy. Powerful mind magier. Gemma Ripley of Andvari is all those things–and determined to stop an enemy from using magical tearstone weapons to conquer her kingdom. Gemma's quest for answers leads her to a trade Summit between the various kingdoms. Among the other royals in attendance is Queen Maeven Morricone of Morta and her son, Prince Leonidas–Gemma's charming and dangerous nemesis. Gemma knows that Maeven always has a long game in motion, and sure enough, the cunning queen invokes an arcane tradition that threatens the fragile truce between Andvari and the other kingdoms. Despite her best intentions, Gemma once again finds herself thrown together with Leo and battling her growing feelings for the enemy prince. When a series of deadly attacks shatters the Summit's peaceful negotiations, Gemma realizes that someone wants to tear the royals down from their thrones–and that this enemy just might succeed.

Categories: Fiction

Rating: 4 stars

Preview

Nettle & Bone

By T. Kingfisher

Published: 26-04-2022

Description: An Instant USA Today & Indie Bestseller An Oprah Daily Top 25 Fantasy Book of 2022 A Vulture Best Fantasy Novel of 2022 An NPR Best Sci Fi, Fantasy, & Speculative Fiction Book of 2022 A Goodreads Best Fantasy Choice Award Nominee From Hugo, Nebula, and Locus award-winning author T. Kingfisher comes an original and subversive fantasy adventure. *A very special hardcover edition, featuring gold foil stamp on the casing and custom endpapers illustrated by the author.* This isn't the kind of fairytale where the princess marries a prince. It's the one where she kills him. Marra never wanted to be a hero. As the shy, convent-raised, third-born daughter, she escaped the traditional fate of princesses, to be married away for the sake of an uncaring throne. But her sister wasn't so fortunate–and after years of silence, Marra is done watching her suffer at the hands of a powerful and abusive prince. Seeking help for her rescue mission, Marra is offered the tools she needs, but only if she can complete three seemingly impossible tasks: -build a dog of bones -sew a cloak of nettles -capture moonlight in a jar But, as is the way in tales of princes and witches, doing the impossible is only the beginning. Hero or not–now joined by a disgraced ex-knight, a reluctant fairy godmother, an enigmatic gravewitch and her fowl familiar–Marra might finally have the courage to save her sister, and topple a throne. "Nettle & Bone is the kind of book that immediately feels like an old friend. Fairytale mythic resonance meets homey pragmatism in this utterly delightful story. It's creepy, funn...

Categories: Fiction

Rating: 4 stars

Preview

An example of the results page, showing recommended books after the user has chosen their filters

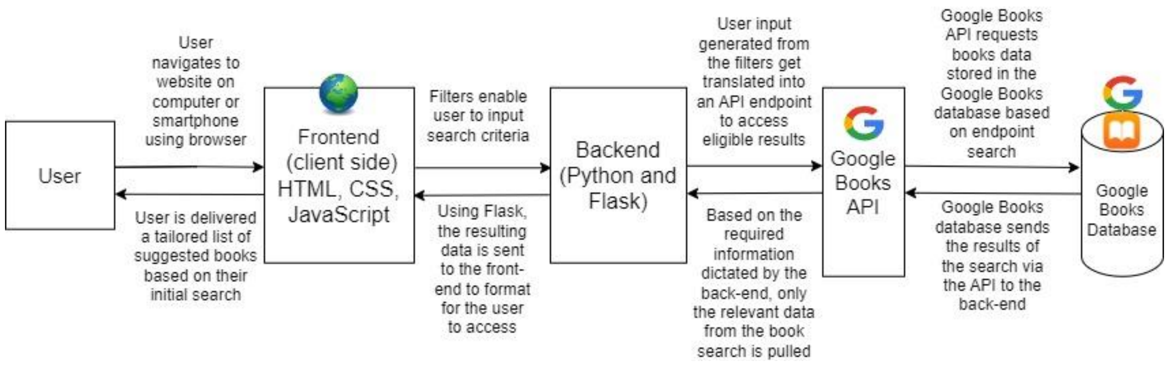
Specifications and Design:

Functional requirements

- A frontend web page that accepts user input via a form made up of input selections to be submitted
- In-built filters coded in the backend: language, age group and rating of 4* & above (later removed, see implementation changes)
- A frontend web page displaying a list of recommended books, decided by the user-input
- A server in order to run the website in the browser (we used Flask for development purposes)

Non-functional requirements

- Optimised delivery of searches (return results in a reasonable amount of time)
- Results returned for all of the possible search combinations
- Well ordered, structured and optimised code for ease of maintenance and repair
- Option to add more filters on the homepage should the user require
- Option to add more information per book on the results page


User
User navigates to website on computer or smartphone using browser
User is delivered a tailored list of suggested books based on their initial search
Frontend (client side) HTML, CSS, JavaScript
Filters enable user to input search criteria
Using Flask, the resulting data is sent to the front-end to format for the user to access
Backend (Python and Flask)
User input generated from the filters get translated into an API endpoint to access eligible results
Based on the required information dictated by the back-end, only the relevant data from the book search is pulled
Google Books API
Google Books API requests books data stored in the Google Books database based on endpoint search
Google Books database sends the results of the search via the API to the back-end
Google Books Database

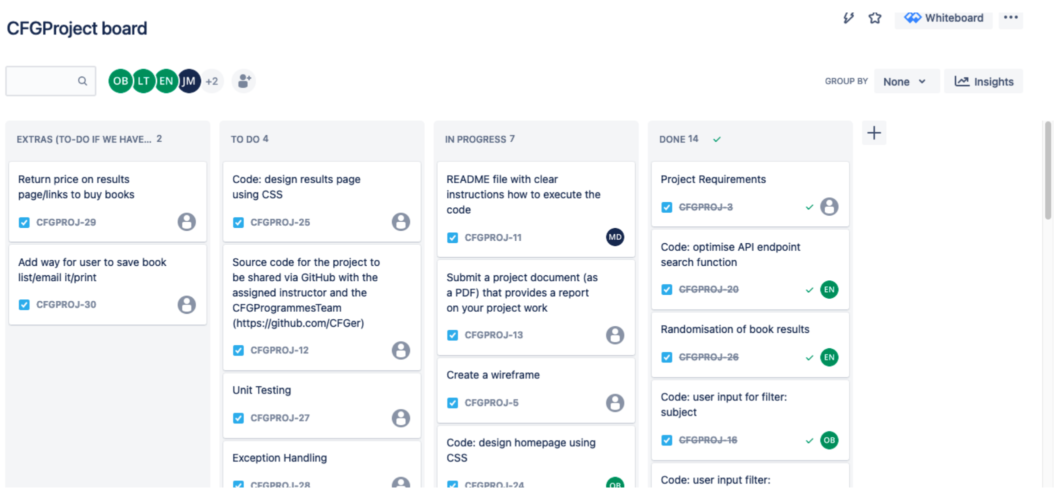
A flow diagram for our website illustrating the system architecture

Implementation and Execution:

Development approach

We approached the project with an agile mindset. We created designated channels of communication; Zoom or Jitsi to call and Slack for sharing updates outside of meetings. All calls were recorded and minutes were taken. We completed weekly sprints, with the longer Thursday session being the time for sprint review, retrospective and planning. We had daily stand-ups (where possible), a mid-sprint review and pair-programming.

We completed code reviews during our meetings; a team member would share their screen and explain what they had written, key functions within the code, what features it implemented, what needed improving etc. This approach meant that all team members understood the code that had been written and could edit on their branch as required.



CFGProject board

Whiteboard

OB LT EN JM +2

GROUP BY None

Insights

EXTRAS (TO-DO IF WE HAVE... 2

Return price on results page/links to buy books
CFGPROJ-29

Add way for user to save book list/email it/print
CFGPROJ-30

TO DO 4

Code: design results page using CSS
CFGPROJ-25

Source code for the project to be shared via GitHub with the assigned instructor and the CFGProgrammesTeam (https://github.com/CFGer)
CFGPROJ-12

Unit Testing
CFGPROJ-27

Exception Handling
CFGPROJ-28

IN PROGRESS 7

README file with clear instructions how to execute the code
CFGPROJ-11 MD

Submit a project document (as a PDF) that provides a report on your project work
CFGPROJ-13

Create a wireframe
CFGPROJ-5

Code: design homepage using CSS
CFGPROJ-24 OB

DONE 14

Project Requirements
~~CFGPROJ-3~~

Code: optimise API endpoint search function
~~CFGPROJ-20~~ EN

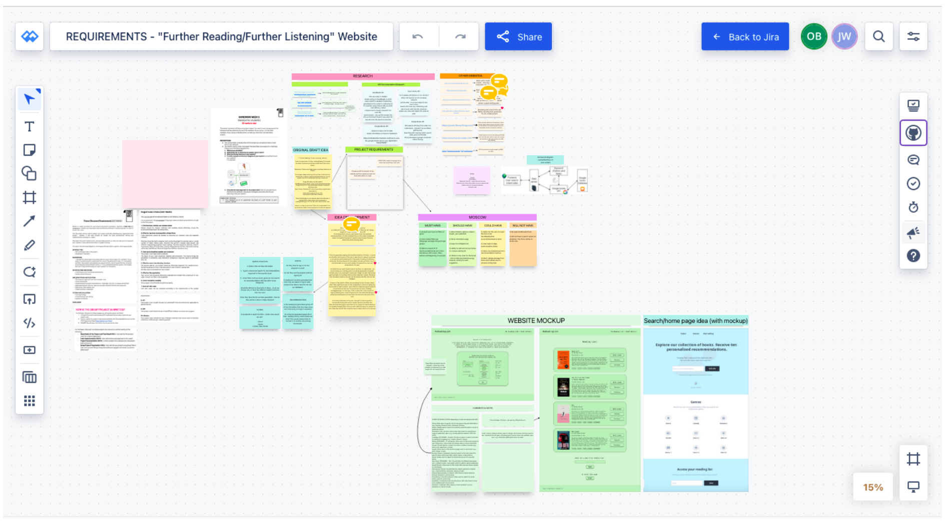
Randomisation of book results
~~CFGPROJ-26~~ EN

Code: user input for filter: subject
~~CFGPROJ-16~~ OB

Code: user input filter:

An image of our Jira Kanban board where we tracked tasks and managed task allocation.

We utilised the Jira Whiteboard feature extensively in the development and design phase. We found the whiteboards useful as they allowed all team members to add ideas simultaneously in a shared space. For example, we used a whiteboard to brainstorm our initial website ideas and to hold a vote on our final website choice and ensure we all had the same vision in mind.


REQUIREMENTS - "Further Reading/Further Listening" Website
Share
Back to Jira
OB
JW
RESEARCH
MOSCOW
WEBSITE MOCKUP
Search/home page idea (with mockup)
15%

Our Jira Whiteboard for project requirements, featuring research, idea development, moscow requirements and website mock-ups (above)

Team-member roles

During our first meeting, we made sure to discuss each team member's availability for the next 4 weeks to establish the workload balance. This ensured we were all on the same page regarding workload expectations and that no-one was given work beyond their capacity. This allowed for a smooth completion of work across the 4 weeks with no tasks unexpectedly needing to be completed. We all contributed to each stage of the process, but roughly the project was divided as follows: Jessica on source code organisation and readability, Ella and Lucy on the backend functions, Millie and Jesse on unit testing and exception handling and Olivia on the front end and documentation.

Implementation Process

The key phases in our development approach were as follows:

Idea development

Discussion of potential ideas, areas of interest, what can be achieved in our time frame.

Planning

Organised implementation of the idea and workload. Decided on the main features vs should/could/won't have features. Agreed which systems we were going to use. Determined priorities and order of approaching tasks. Mock-up created using Figma for our website design.

Backend coding

Began with implementation of major components e.g. API, Flask. Then moved into smaller tasks e.g. coding helper functions.

Frontend coding

Creation of HTML files, with use of JavaScript for dynamic elements, to create the user-facing website. Addition of CSS for styling and presentation of the website.

Testing

Completed user-input testing to catch errors. Debugged the program, implemented Exception Handling and completed unit testing.

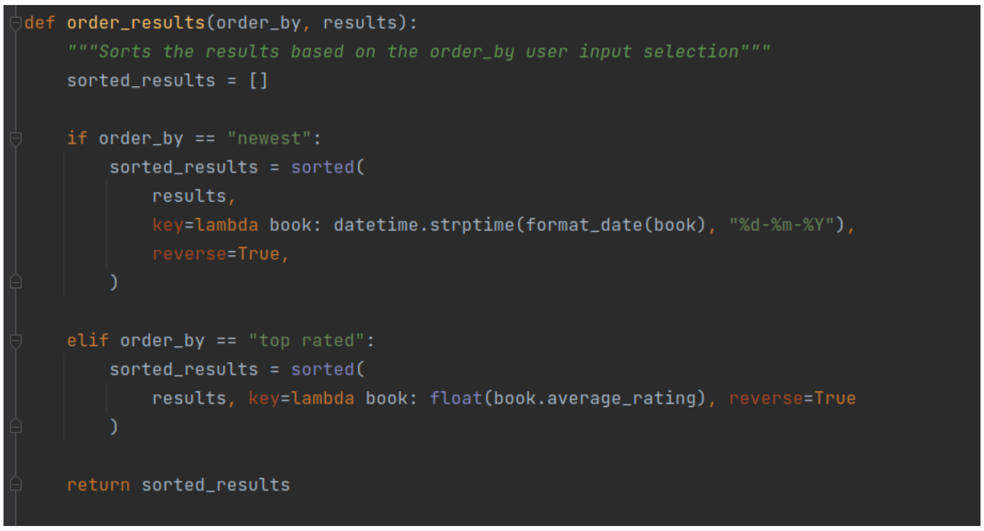
Refinement

Ensured good coding practices had been followed throughout all files in line with SOLID principles. Ensured no code is repeated, that we had clear code with good commenting throughout. Created the README with thorough instructions for how to run the program for users.

Tools and libraries

- **Google Books API:** we chose this API as the GoodReads API was no longer offering new keys and the OpenLibrary API didn't offer the same level of search criteria
- **GitHub**: used for version control - we worked on separate branches for new features, created pull requests to review and merged to the main frequently
- **Flask**: to run as a development server
- **Requests Library**: to access and extract data from the API using a dynamic endpoint
- **Python 3.11:** for coding the backend of our website
- **JavaScript:** for coding the dynamic element of our frontend
- **HTML:** for coding the frontend of our website
- **CSS**: for styling our frontend
- **IDEs**: PyCharm and VS Code
- **Insomnia**: for testing the endpoint in order to optimise our searches

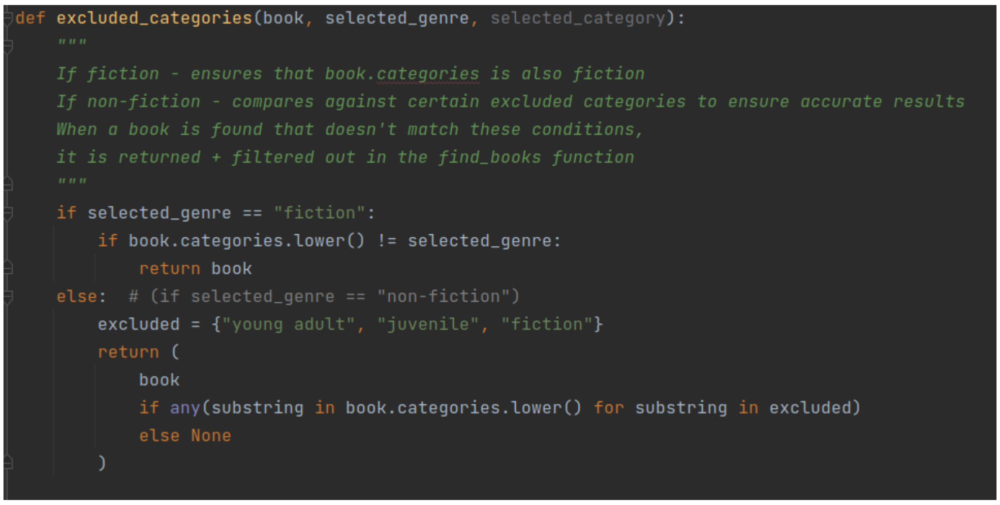
Implementation changes & challenges:
One of our greatest challenges was harnessing the data available from the Google Books API as it became evident that it was not normalised and could not be relied upon for extremely narrow searches. While we initially set out to filter our searches based on genre, category, book length, published date and price - all with a minimum rating of 4 stars - we soon found that the overlap of filters was causing our search results to be extremely limited. Therefore, we looked at each of the filters and how we might alter them for a better product.



```python
def order_results(order_by, results):
    """Sorts the results based on the order_by user input selection"""
    sorted_results = []

    if order_by == "newest":
        sorted_results = sorted(
            results,
            key=lambda book: datetime.strptime(format_date(book), "%d-%m-%Y"),
            reverse=True,
        )

    elif order_by == "top rated":
        sorted_results = sorted(
            results, key=lambda book: float(book.average_rating), reverse=True
        )

    return sorted_results
```

Python coded filter to sort the results in descending order by date or rating

First, we removed the option to filter by price - we had to remove this attribute from the book class as with so many of the entries having no price associated with them, they were returning a "N/A" value which our find_books() function would discard. Once we removed this attribute, we immediately returned more results. During the process of coding the filters, we also found ourselves asking questions about their relevance. For instance, whilst styling the year range filter, we realised that the publication date was somewhat irrelevant as publication dates change with edition e.g. you don't need to filter publication year to 1597 to return Romeo and Juliet. This was removed as a choice filter and reintroduced as a sorting filter.



```python
def excluded_categories(book, selected_genre, selected_category):
    """
    If fiction - ensures that book.categories is also fiction
    If non-fiction - compares against certain excluded categories to ensure accurate results
    When a book is found that doesn't match these conditions,
    it is returned + filtered out in the find_books function
    """
    if selected_genre == "fiction":
        if book.categories.lower() != selected_genre:
            return book
    else:  # (if selected_genre == "non-fiction")
        excluded = {"young adult", "juvenile", "fiction"}
        return (
            book
            if any(substring in book.categories.lower() for substring in excluded)
            else None
        )
```

Function coded in Python - a filter to better exclude unrelated results

We thus introduced two 'sort by' filters: 'rating' and 'newest'. By allowing the user to choose between these two options, we significantly reduced the overlapping filters and could return books from all publish years and all ratings, whilst still being able to use this data to format the list of results and return the information in a useful way. Optimising the filters in this way greatly improved the speed of the program, the quality and number of results returned across all categories, whilst ensuring that the work we had already done creating the functions and processing the data was not wasted by discarding the filters entirely.

As the Google Books API only returned 40 books per call, we had to use pagination to ensure that our function kept calling until we had 10 results. On occasions when the search couldn't find 10 books, our function would keep increasing the start index creating an infinite loop. Due to this, we had to introduce a variable to keep track of how many times 0 results had been found on a page, and break the loop after 100 pages when it was likely no more results would be found. This meant we sometimes had less than 10 books for particular searches, but it improved the speed of the program and ensured some books were returned, rather than having none and an infinite loop.

Whilst working out the best query parameters for the API endpoint, we found that the addition of extra search terms often improved or worsened the results. By utilising Insomnia, we were able to run different searches to see how many results were returned with each endpoint and what the major categories were. We then used this information to write functions that would format the category for the API call, e.g. if the genre is 'fiction' the query parameters would be "q=fiction+(selected_category)", whereas 'non-fiction' would be "q=subject:(selected_category)". If any of the categories were two words such as 'Science Fiction', we found removing the space and searching as one word also really improved the results.

In an attempt to increase the variety of books returned, we set both the page number and number of pages to add at the end of each loop to a randint in the fetch_books function of the API call. Although this somewhat improved the variety of results returned, we eventually made the decision to remove this randomisation. Because the page number was generated randomly, it led to non-sequential page numbers meaning that the API endpoint was bouncing backwards and forwards which slowed down the call. We often still saw a repetition of books, as despite it being random it couldn't ensure that the same book doesn't appear twice in a row. Therefore it didn't seem to serve its purpose and we agreed focusing on getting solid results for each different search was more important.

The last thing we noticed was the occasional result that didn't fit into the filters we had selected. Therefore, we created a function to exclude particular book categories from appearing. From the beginning, our target audience was 18+, so we added words such as 'children's' and 'juvenile' to an excluded categories list which was compared to the category in the book class. For non-fiction we ensured that no fiction categories would be returned, and for fiction we ensured that only fiction categories were returned, and this helped to normalise our results.

Testing and Evaluation:

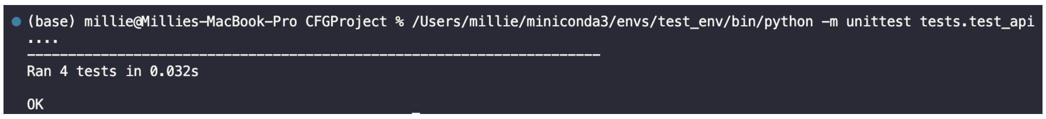
Exception handling / debugging

We implemented exception handling and debugging in our code using try/except blocks and print statements. This allowed us to handle errors gracefully, such as invalid user inputs, data formatting issues and API errors. To aid in debugging, we structured our code into smaller modular functions that each had one problem to solve, keeping it simple. Custom error messages, rather than generic ones, were printed and comments and docstrings were added as much as possible to help debug issues.

Unit testing

We wrote unit tests for critical program functions using the *unittest* module in Python. We tested functions in isolation to verify they performed as intended. Some examples:

- Tested our call API functions to ensure they properly
- Wrote tests to ensure our book filters filtered as expected



```
(base) millie@Millies-MacBook-Pro CFGProject % /Users/millie/miniconda3/envs/test_env/bin/python -m unittest tests.test_api
....
----------------------------------------------------------------------
Ran 4 tests in 0.032s

OK
```

Example of successful unit tests on the API call

Functional and user testing
After we edited the final search criteria and 'order by' categories on the home page, we conducted testing of all possible user inputs. We split the combinations between us to test all possibilities. During this process, we encountered a ValueError relating to the date-time. We also encountered a Flask error appearing in the console when the program ran. We were able to discover and fix these errors as a result of our thorough user testing approach.

Evaluation/review of code
One of the final steps of our project was checking the quality of our code and that it abided by industry standards. We ensured that the code abided by SOLID principles, that there was no repetition of code and that each function was doing only one thing. We also organised our files effectively and constructed clear instructions for the ReadMe so that the program would be clear for anyone to use.

Conclusion

Successes:
We successfully built a program that matched our core proposal: a website to generate book recommendations based upon user input filters. Each of our 'must have' features was completed. In addition, we were able to implement some extra features, such as a 'print' feature for the user to save their list and a 'preview' feature which provided a link for the user to see and purchase each book. One of our key successes was our team approach. We completed the project on time and exceeded our MVP due to our agile approach and team organisation.

Improvements - scalability of the project in the future
If we had more time, we would have liked to improve our website further by:
- Creating a way for a user to store book recommendations and refresh to get new results that were not previously given
- Expand the search criteria so that a user could type in any subject/topic if they wanted, rather than use set fiction and non-fiction categories
- Caching API requests to reduce overhead and optimise performance

What we have learnt:
Knowing what we know now, we would have done the following differently:
- Found out if there was a way of scraping the data from the API to better analyse and understand as we were finding that it was uneven and poorly organised
- Refined our GitHub workflow earlier on to avoid so many merge conflicts
- Assigned Jira tasks more quickly so that we could start coding sooner