Introduction

In this capstone, you will create an application called TomeRater that allows users to read and rate books.

The purpose of this capstone is to practice implementing and testing classes in Python. You will be given specifications for methods for 5 different classes that interact with each other. To create these methods you will use your knowledge of lists, loops, dictionaries, strings, control flow, and of course, basic Python syntax! If you need to refresh your memory of these concepts, go back to these lessons in the syllabus.

Compared to the other projects you have completed this far, we are requiring few restrictions on how you structure your code. In addition, much of the code you write for later parts of this project will depend on how you decided to implement earlier parts. **Therefore, we strongly encourage you to read through the entire assignment before writing any code.**

~~Create a User~~

We will need to keep track of our users. We have provided a skeleton for a User object in **TomeRater.py**. Open this file in your text editor. We are going to replace the `pass` keywords you see, so that these methods work. The User class should have:

- ~~A constructor method, which takes in `self`, `name`, and `email`. It~~

~~should set instance variables `self.name`, `self.email`~~, ~~and `self.books`~~.

    - ~~`name` will be a string~~
    - ~~`email` will be a string~~
    - ~~`self.books` is an empty dictionary that will map a Book object (which we will create!) to this user's rating of the book~~

- ~~A method `get_email` that returns the email associated with this user~~
- ~~A method `change_email` that takes in a new email and changes the email associated with this user. It should also print a message saying that this user's email has been updated.~~
- ~~A `__repr__` method that returns a string to print out this user object in a meaningful way. Printing a user named Stephen Hawking, with an email hawking@universe.edu, with 7 books read, might produce a string like~~:

```
User Stephen Hawking, email: hawking@universe.edu, books read
: 7
```

but you can create your own format for a string that is most helpful for you to understand the user object being printed.

- ~~An `__eq__` method to define comparison between users. A User object should be equal to another User object if they both have the same name and email.~~

Create a Book

Our User objects will be reading books. But what is a book? Define a Book object that has:

- ~~A constructor method, which takes in `self`, `title`, and `isbn`. It should set instance variables `self.title` and `self.isbn`. It should also set an instance variable `self.ratings`, which will start as an empty list.~~
    - ~~`title` will be a string~~
    - ~~`isbn` will be a number~~
- ~~A method `get_title` that returns the title of the book.~~
- ~~A method `get_isbn` that returns the ISBN of the book~~
- ~~A method `set_isbn` that takes in a new ISBN and sets the book's `isbn` to be this new number. It should also print a message saying that this book's ISBN has been updated.~~
- ~~A method called `add_rating` that takes in a `rating` and adds it to the list `self.ratings`. It should only do this if `rating` is a valid rating (at least `0` and at most `4`). Otherwise, it should print `"Invalid Rating"`.~~
- ~~An `__eq__` method to define comparison between books. A Book object should be equal to another Book object if they both have the same title and isbn.~~

Make a Fiction Subclass of Book

Books are not all created equal. Let's assume we have two kinds of books, fiction and non-fiction.

The `Fiction` class should inherit from Book and have the following methods:

- ~~A constructor, which takes in `self`, `title`, `author`, and `isbn`. It should first call the `__init__` of its parent class, with `title` and `isbn`. Then, it should set the instance variable `self.author`~~.
- ~~`get_author`, which returns the author~~
- ~~`__repr__`, which will return the string:~~

```
{title} by {author}
```

For example, the book with title “Alice In Wonderland” and author “Lewis Carroll” would print:

```
Alice In Wonderland by Lewis Carroll
```

Make a Non-Fiction Subclass of Book

The `Non_Fiction` class should inherit from Book and have the following methods:

- ~~A constructor, which takes in `self`, `title`, `subject`, `level` and `isbn`. It should first call the `__init__` of its parent class, with~~

~~`title` and `isbn`. Then, it should set the instance variables `self.subject` and `self.level`.~~
    - ~~`subject` will be a string, like `"Geology"`~~
    - ~~`level` will be a string, like `"advanced"`~~
- ~~`get_subject`, which returns the subject~~
- ~~`get_level`, which returns the level~~
- ~~`__repr__`, which will return the string:~~

```
{title}, a {level} manual on {subject}
```

For example, the book with title "Society of Mind" about beginner Artificial Intelligence would print out:

```
Society of Mind, a beginner manual on Artificial Intelligence
```

Give Books and Users Methods

Now that we have both Book classes and User classes, we can create more methods than just ones that get and set instance variables.

For your User class, add two methods:

- ~~`read_book`, which takes in `book` and an optional parameter `rating`, which defaults to `None`. It should add a key:value pair to `self.books` where the key is `book` and the value is `rating`.~~
- ~~`get_average_rating`, which iterates through all of the values in~~

~~`self.books`, which are the ratings, and calculates the average rating. It should return this average.~~

For your Book class, add the method:

- ~~`get_average_rating`, which iterates through all of the values in `self.ratings` and calculates the average rating. It should return this average.~~

There is one more method we have to add to Book to make this work! Do you remember how we get the error `TypeError: unhashable type: 'list'`, when we try to create a dictionary with lists as keys? This is because lists are mutable, and thus do not have a consistent *hash* that a dictionary can use to look up the associated value. We are trying to make a dictionary in the User class called `self.books` that has Book objects as keys. In order to use a class that we construct ourselves, we must make sure that our object is `hashable`, and not `unhashable`, like a list!
To make our Book hashable, we will add a method `__hash__` which will return a consistent hash for an instance of a book object:

```
def __hash__(self):
  return hash((self.title, self.isbn))
```

Copy this method into your Book class so that Book becomes hashable! If you're curious, look at the documentation to see what the built-in method `hash()` is doing: https://docs.python.org/3/library/functions.html#hash

Create TomeRater

We have Users and Books, but how do they interact? Now it's time to create the application to store those users. It is time to create TomeRater!

Create a class `TomeRater` that has the following methods:

- ~~A constructor that only takes in `self`. It should create:~~
  - ~~`self.users`, an empty dictionary that will map a user's email to the corresponding User object~~
  - ~~`self.books`, an empty dictionary that will map a Book object to the number of Users that have read it~~
- ~~`create_book`, which takes in `title` and `isbn` and creates a new book with that title and ISBN. Returns this Book object.~~
- ~~`create_novel`, which takes in `title`, `author`, and `isbn`, and creates a new Fiction with that title, author and ISBN. Returns this Fiction object.~~
- ~~`create_non_fiction`, which takes in `title`, `subject`, `level`, and `isbn`, and creates a new Non_Fiction with that title, author and ISBN. Returns this Non_Fiction object.~~
- `add_book_to_user`, which takes in `book`, `email`, and an optional parameter `rating`, which defaults to `None`. It should get the user in `self.users` with the key `email`. If this user doesn't exist, it should print out `"No user with email {email}!". If the user exists, it should:
  - ~~Call `read_book` on this user, with `book` and `rating`~~
  - ~~Call `add_rating` on `book`, with `rating`~~

    - ~~Check if the book is in TomeRater's `self.books` already. If it is not, add the key `book` to `self.books` with a value of 1 (because one user has read it)~~
    - ~~If `book` was already in the catalog, increase the value of it in `self.books` by 1, because one more user has read it.~~
- `add_user`, which takes in `name`, `email`, and an optional list of Books `user_books` that defaults to `None`. It should create a new User object from `name` and `email`. Then, if `user_books` is provided, it should loop through the list, and add each Book to the user (using the TomeRater method `add_book_to_user`)

Create Some Analysis Methods for TomeRater

Now, we have an application that keeps track of Users and the Books they have read. We should add some methods to be able to analyze our collection.

Add these methods to the `TomeRater` class:

- ~~`print_catalog`, which iterates through all of the keys in `self.books` (which are Book objects), and prints them~~
- ~~`print_users`, which iterates through all of the values of `self.users` (which are the User objects), and prints them~~
- ~~`most_read_book`, which should iterate through all of the books in `self.books` and return the book that has been read the most. Remember that the keys of `self.books` are Books, and the values are how many times they've been read.~~

- ~~`highest_rated_book`, which should iterate through all of the books in `self.books` and return the book that has the highest average rating. Remember that the keys of `self.books` are Books, and you can call `book.get_average_rating()` on a Book object `book`~~.
- ~~`most_positive_user`, which should iterate through all of the users in `self.users` and return the user that has the highest average rating. Remember that the values of `self.users` are Users, and you can call `user.get_average_rating()` on a User object `user`~~.

Do Some Analysis

Open your command line and navigate to the *TomeRater* directory:

```
cd TomeRater
```

Run the file *populate.py*, using the command:

```
python3 -i populate.py
```

This file, **populate.py**, will create a TomeRater object called `Tome_Rater` and fill it with some users and books. The `-i` flag will leave the terminal open for you to play with. You can then run some of your methods you created in the last section by using syntax like:

```
Tome_Rater.most_positive_user()
```

Go ahead and open **populate.py** in your text editor as well and look at the objects that are being created and used. Add your own tests, to see if your application works the way you expect it to!

Get Creative!

Incredible! You have an application that stores users and books! You basically own a startup now. To take your project to the next level, choose one of the following extension ideas to implement:

- Add more sophisticated error testing! This could include:
    - ~~If someone tries to add a user with an email that already exists in `TomeRater`, print out a message telling them that this user already exists~~
    - ~~Make sure that books all have unique ISBNs~~
    - ~~Make sure that an email address is valid by checking if it has an `@` character and either `.com`, `.edu`, `.org`~~
- Add some *dunder methods* to the `TomeRater` class as well:
    - ~~What should it look like if a `TomeRater` object is printed?~~
    - ~~What would it mean for two `TomeRater` objects to be equal?~~
- Add more sophisticated analysis methods like:
    - ~~`get_n_most_read_books`, which would take in a number `n` and return the `n` books that have been read most, in descending order~~
    - ~~`get_n_most_prolific_readers`, which would take in a number `n` and return the `n` users that have read the most~~

    books, in descending order

- Add a price variable for each Book so that you could add methods to TomeRater like:
    - ~~`get_n_most_expensive_books(self, n)`, which would return the `n` books with highest price~~
    - ~~`get_worth_of_user(self, user_email)`, which would return the sum of the costs of all the books read by this user~~
- ~~Your own idea for how to make TomeRater better~~