ECE 356 Design Project

Repository Link: https://github.com/dalenu/356project
Video Link: 356 Video Link

Ideal Design, Issues Faced and Subsequent Implementation

We will start off with discussing what we had hoped to accomplish initially when we were assigned this data, then discuss what measures we took to implement this, or issues we faced that prevented us from implementing this the way we had envisioned.

Checkout Records and Checking Out/In a Book

Our vision with these records was to use them to track the availability of books across the library. This way if a user wants to check out a book in the current day we can easily execute it if the book was indeed available. We would simply iterate over the entries with the book's BibNumber from the records, and then decrement the item count by 1 every time it was checked out. That way we can determine how many copies we had of this book at the end of the month.

There are a few major issues we faced with this that practically forced us to change the role these records would play in our implementation. For starters, to check the current item count, we would have to cross reference that with the library's inventory - which is great in theory. The issue is that the inventory was stocked at only 2 dates; September and October and ONLY on the first of those months, while the checkout records end abruptly on October 18th. This poses some uncertainty in the execution, since the checkout records end in the middle of the month. We only know at the start of the month that we have a certain number of copies; but then what happens at the start of the next month (November) when the records end on the 18th of October? What happens when someone checks it out on the 18th; do we just mark the book as unavailable? How do we know it was ever returned?

To illustrate another issue, consider the case where the records state that Book A was checked out on August 10th, we can not check the item count of this book in the inventory since we only know the number of copies available on October 1st; with no clue how many existed in August. Another issue we had was the fact that there were several entries per BibNumber for each (location, type, collection) tuple. This means that two Books can have the exact same BibNumber, Type, and Collection - with only the Location to separate the two entries. The issue is that the records do not show us the location this book was checked out at. What this means is that every time someone checks out a book, we would have to query the 2 million+ CSV, choose a location at random (since it was never specified, meaning we can potentially have multiple of the same entry), and then decrement its count by 1. The fact that location was absent also restricts the client application's potential; Users should be able to check in and checkout a book to/form a location they please. Not tracking the location in the records means that we don't give the User that choice.

What we realized was that we practically can not consider the checkouts before October as relevant - we will just assume that everything in the October inventory is available and check in/out based on that item's count specifically; the checkout records for the earlier part of the year will be kept for data mining or for querying purposes. Any time a User wants to check out a book, we will simply ask for the itemType, itemCollection, branch and BibNumber of the book; which gives us a singular entry in the inventory and allows us to easily change the itemCount as we please.

You are probably thinking “how would a User know all this information about a book?” Not to worry; we have a function that takes ANY part of a title and outputs a CSV with all the matching titles and their subsequent information, so you will always have a way of knowing the identifying information of any book in the library. For any new checkouts, we will store that in a table containing the libraryCard of the User who checked it out, and the location it was checked out from in addition to the aforementioned identifiers. Checking in a book just simply edits the check in date, and updates the status of the book.

To summarize, there are currently 2 versions of the checkout records - new and old. The old ones are not too helpful in the sense that they give us only book-specific information; regardless of the branch the book was checked out at and the User that checked it out. There are limitations to this, since a Branch can not see which books were checked out from its branch, and a User can not see which books they checked out; nor can an Admin track which books are overdue for a User, etc. As a result we have 1 table for old records; just for the sake of book keeping and preserving records; and then we have a new table for all Users after October 1st, with more comprehensive data stored in this table.

ISBN and GoodReads Ratings
This is by far the most frustrating aspect of this data; one that was most detrimental to our ideal execution of this project. In theory, the ISBN of a book in 1 database should be able to match it to another database such as GoodReads and return the rating etc. Except not exactly; since for some reason nearly every book in Seattle’s library has more than 1 ISBN. Which of these do we choose? Do we take the first ISBN or try and loop through all ISBNS to check for a match in GoodReads? Do we discard the ISBN’s that do not match? You can imagine how ridiculously inefficient it is to iterate over 1 million+ books and cross reference them with the GoodReads database that’s nearly 4 times larger. Another issue is the fact that there are just so many different ISBN for the same book title, and many others do not even have an ISBN, and many other just simply do not exist in the GoodReads data set to the point where it is useless if you want to use it as an identifier across both datasets. What we opted to do instead was just drop all null ISBN from GoodReads, and make the (Title, ISBN) tuple a primary key in GoodReads - to allow us to have entries that have duplicate Titles, just different ISBNs. We then tracked every single BibNumber with multiple ISBN, and wrote a script that creates a new entry for each (BibNumber, ISBN) pair - and then we inner join on the ISBN column with GoodReads. The result is a Foriegn Key that relates the Book BibNumber to this Temp table’s BibNumber; and a Foriegn Key from Temp’s BibNumber to GoodReads ISBN. This allows us to relate the Books in our Seattle Library to the GoodReads Reviews.

Titles in Seattle/GoodReads and GoodReads Ratings
Another idea we had was to match the titles across both datasets; but that proved futile. The way titles are formatted in Seattle right now is as follows: “Title of Book / authors and all contributors”, in contrast to GoodReads lack of consistency. Take these 2 entries for example:

110938,The Awakening and Other Stories,0613708458,Kate Chopin,3.86,2000,14,11,Turtleback,5:239,4:255,3:196,2:57,1:13,total:760,2,,375,,

110939,The Awakening & Selected Stories (Modern Library),0679424695,Kate Chopin,3.92,2000,1,11,Modern Library,5:3793,4:3276,3:2139,2:743,1:342,total:10293,3,,354,,

If we filter out the database to remove the brackets, then we get 2 books with the exam title - which of these ratings do we choose? GoodReads has multiple titles of the same book but added extra identifiers to the point where it is practically impossible to get just single unique entries in the dataset.

Filtering Data

Checkout Records
There were millions of entries in the 2017 csv alone; let alone the ones from 2005-2016. In fact, there are nearly 5 million just from January to September; which is just pointless data to store since we had no Data Mining component. We opted to just use the records from August to October, which has 500K entries on its (as mentioned in piazza) and allow the User to query those for Data Mining if need be. We took all the attributes from these records.

Seattle Inventory
We chose to only keep the latest version of the inventory, which are the entries with the report date “01/10/2017”; anything else was dropped since we need the most recent inventory to ensure we are executing check in/out queries correctly. All entries with empty BibNumbers/ItemType/Title/Collection/Location were also dropped. Each book has a branch, a type and a collection. The issue with these values is that they are basic string values; what this means is that any time we want to change the identifier of collection/branch/type, we will have to query the entire dataset and implement that change. Instead, what we did was map an integer to each identifier and store that into a table (id, value) to prevent such a case from happening.

GoodReads
We took 2 million + entries from GoodReads, dropping only anything that doesn’t have an ISBN. We also noticed some csv’s had more/less columns then the others, so we normalized that data by adding those rows.

Summary Of Implementation:

Rating and Reviewing within a Library
We idealized that Users in a library should be able to rate and review Books they have in their own library - to inspire a sense of community amongst the Users. In the end, we implemented this in a way such that there is a foriegn key from User to Review, and a Foriegn key from

Review to Book; so that all things are connected; a User can Review a Book, and Check in/out a book. This implementation allows the User to edit their old reviews as well; and now multiple Users can rate and review a Book as well.

Finding Ratings from GoodReads
Our final implementation just asks for the title the User is curious about, and then we will "select where" using regexp on the Title in the Seattle Library, and then finally search for that ISBN in GoodReads. This means that all a user really needs is just the Title of a Book; which is a realistic scenario.

Checking In and Checking Out a Book:
When a User checks out a Book, we create an entry in the Book table (BibNumber, CheckOutDate, CheckInDate, DueDate, Status, Type, Collection, Branch) where the status is 0 if the User has just checked it out. The due date is just 2 weeks ahead of the check out date. When a User returns the book, we check if the return date was > 2 weeks from the check in date; if so then we mark it as a 2 (overdue) or a 1(on time) A User can also not checkout the same book more than once in the same 24 hours.

Searching for Book
As it stands, we have allowed the User to search for a book by Title, or search for it by either ItemType, ItemCollection or both. The output of any of these returns a CSV with the information (Bib, Author, Title, etc) into a CSV, to avoid congesting the client console. We also have non-client facing functions, such as searching for the ISBN of a book by its BibNumber to help us with our queries.

Checkout Records
The old ones are not too helpful in the sense that they give us only book-specific information; regardless of the branch the book was checked out at and the User that checked it out. There are limitations to this, since a Branch can not see which books were checked out from its branch, and a User can not see which books they checked out; nor can an Admin track which books are overdue for a User, etc. As a result we have 1 table for old records; just for the sake of book keeping and preserving records; and then we have a new table for all Users after October 1st, with more comprehensive data stored in this table. A User can also see how many active checkouts they have currently. Apart from the count and searching for a rating in GoodReads, the result of these searches is always outputted to a CSV

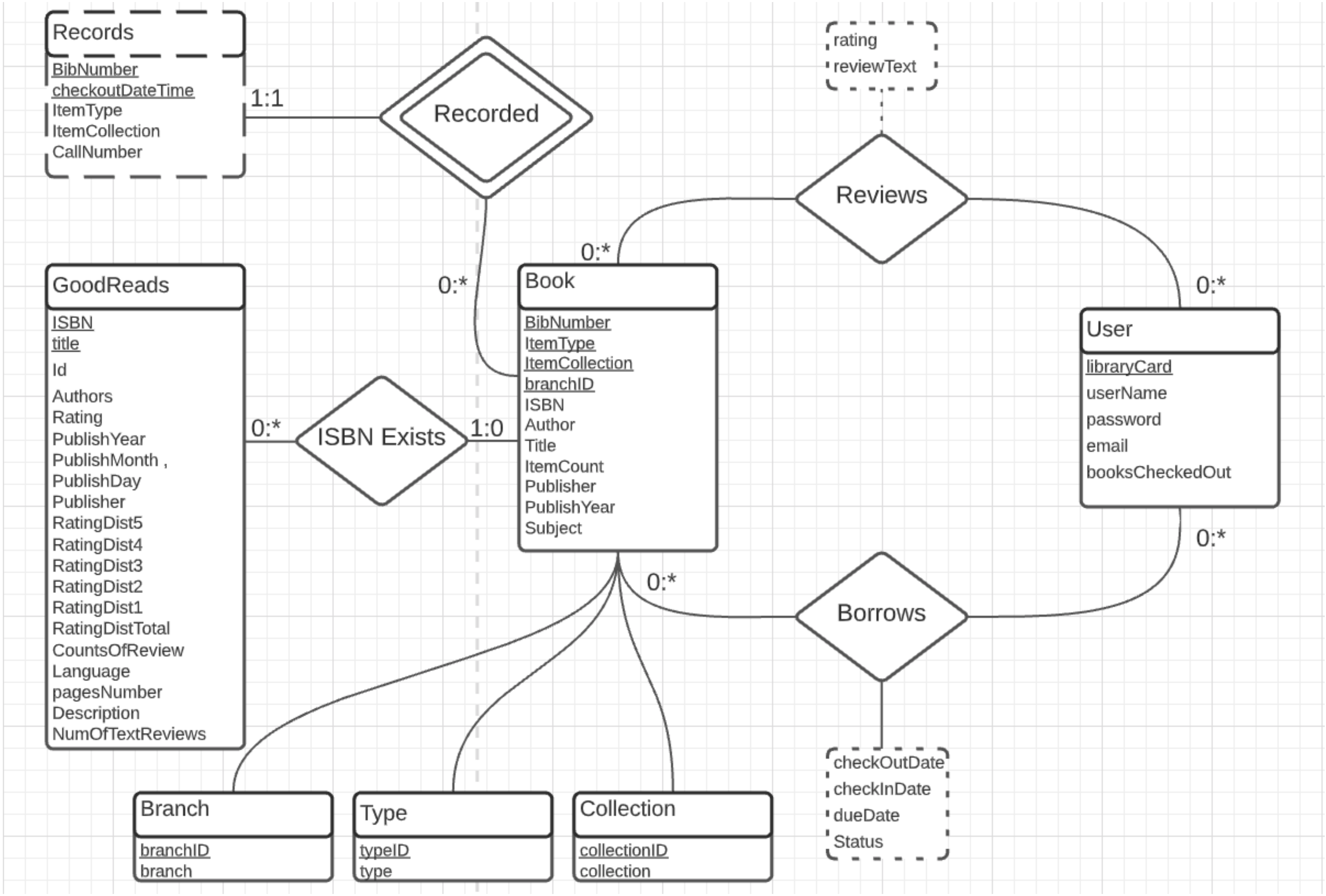
Admin
We have allowed administrators a host of functionality; adding/removing a Book/User/Review, as well as seeing which/how many books a User has checked out.

Functionality

1. Create a User
2. Add and Remove a Book
3. Add, Edit and Remove Seattle Library Reviews

4. Find GoodReads Rating for Book
5. Find Book Info by Title
6. Find Books based on Category
7. Check In and Checkout a Book
8. Find all Records for a Books by BibNumber
9. Find all Records for User by libraryCard
10. Find Number of Books Currently Checked out by User

ER Diagram



The diagram above illustrates the relationships in our project. A User can Borrow as many books as he wants (0 : *) and a book can be Borrowed by as many Users as exists, or even 0 Users. Likewise, a User can Review a book several times, or however many Books he wants, while a Book can have 0 reviews about it, or as many Reviews as there are Users in the database. Each book has a Branch, Type and Collection as specified. Lastly, if the ISBN exists in the GoodReads database, the Book will have a singular entry for it, while it may also not exist

(0:1). At the same time, GoodReads data can have 0 books from Seattle Library in its database, or as many Books as exists in the Library. Records is a weak entity relying on Book, represented by the relationship Recorded. It represented grandfathered data from the original Seattle Library that seemed unrealistic to process and was not helpful for a user's ability to check out books. However, we kept the records for general bookkeeping. The Branch, Type and Collection were each stored on their own as mentioned earlier; string identifiers are not ideal since they are liable to change in the future and so we mapped each string to a unique integer instead, and stored that tuple in a table. There are several possible changes we could have made here, starting with the multivalued ISBN in Book. Usually these values would be stored in a separate table, but we already do that when we inner join with GoodReads - we just have this here to show that the ISBN is not neglected. The RatingDist in GoodReads could possibly be their own relationship, but it does not really help us much since unlike Type, Collection, and Branch, these values are not duplicated across the table - it is highly unlikely that a 2 separate entries will have the exact same rating; but it is still more correct to have them in their own table although it would definitely be more inefficient since we would have to change our filtering, loading and queries to adapt to this change. The same holds for the publication related information - what are the chances that two different books were published on the same day, month, year and by the same publisher? The data is almost never duplicated so adding it in a separate table will just mean that each review entry has one publisher related entry which is really not any different than having them in the same table. What we are trying to say here is that these are not multivalued attributes; we can definitely agree it is more organized to store them in their own entities but those would be weak entities and would also come at a cost in the form of more sophisticated querying for little return. You can argue that the Records entity is not a weak entity but in this case it is because these entries are not unique to a specific Book; the same Book can be checked out at the same time in these records since they do not specify the location; meaning the same BibNumber can exist in the same record with the same data (assuming only the location is different between the two entries) In that sense, these records heavily depend on the existence of the BibNumber of the Book for their Primary Key and the records would be meaningless without them. We had to ignore what the CallNumber and Barcode did here because they were never explained, but we included them in our data for completeness.

DataBase Design

Each book has a branch, a type and a collection. The issue with these values is that they are basic string values; what this means is that any time we want to change the identifier of collection/branch/type, we will have to query the entire dataset and implement that change. Instead, what we did was map an integer to each identifier and store that into a table (id, value)to prevent such a case from happening.

```
create table Branches (
   branchID INTEGER PRIMARY KEY,
   branch VARCHAR(5) NOT NULL
);

create table ItemCollection(
   collectionID INTEGER PRIMARY KEY,
   collection TEXT NOT NULL
);

create table ItemType(
   typeID INTEGER PRIMARY KEY,
   type TEXT NOT NULL
);
```

The User table is quite simple; each User has a password, email, User name you would expect of any client that is relational. In addition each User has alibraryCard since that is what we will use to track the checkouts and check ins. The last attribute the User has is the booksChecked out, which is just a count of all active checkouts.

```
create table User (
   userName TEXT NOT NULL,
   password TEXT NOT NULL,
   email TEXT NOT NULL,
   libraryCard INTEGER PRIMARY KEY,
   booksCheckedOut INTEGER NOT NULL
);
```

Each Book in our database will have a 4-tuple as its unique identifier (BibNumber, branchId, ItemCollection, ItemType) Additional attributes have been added such as the Author and year the Book was published, or what subject it is related to. The most important of these additional attributes is the ItemCount, which allows us to implement our logic for checking in/out a Book. Given that a book has a Branch, Type and Collection, each of these is a Foreign Key to their respective tables.

```
create table Book (
   BibNumber INTEGER NOT NULL,
   Title TEXT NOT NULL,
   Author TEXT,
   ISBN TEXT NOT NULL,
   Publisher TEXT,
   PublishYear TEXT,
   ItemType INTEGER NOT NULL,
   Subject TEXT,
   ItemCollection INTEGER NOT NULL,
   branchID INTEGER NOT NULL,
   ItemCount INTEGER NOT NULL,
   PRIMARY KEY (BibNumber, branchID, ItemCollection, ItemType),
   FOREIGN KEY (branchID) references Branches(branchID),
   FOREIGN KEY (ItemType) references ItemType(typeID),
   FOREIGN KEY (ItemCollection) references ItemCollection(collectionID)
);
```

We mentioned earlier that we allow a User to Review a Book. This here is the table that will embody this dynamic. The review is for a specific Book, by a specific User which is why it must contain the BibNumber of a Book and the libraryCard of the User; this tuple will be the primary key. Each review has a rating (decimal) and a review of how the User felt, ex: ("great book, loved the plot!"). The BibNumber is a foriegn key to Book, while libraryCard is a Forigen Key to User

```
create table LibraryReview (
   BibNumber INTEGER NOT NULL,
   rating DECIMAL(2,1) NOT NULL,
   reviewText TEXT NOT NULL,
   libraryCard INTEGER,
   PRIMARY KEY (BibNumber, libraryCard),
   FOREIGN KEY (libraryCard) references User(libraryCard)
   FOREIGN KEY (BibNumber references Book(BibNumber)
);
```

Since this is a Table that consists of Books that have been checked out, it makes sense for the BibNumber to be included in the Primary Key; in addition to the checkoutDateTime. Each entry has a Type and Collection, so those Foreign Keys have also been added. The only additional attribute is just the CallNumber.

```
create table Records(
   BibNumber INTEGER NOT NULL,
   ItemType INTEGER,
   ItemCollection INTEGER,
   CallNumber TEXT,
   CheckoutDateTime datetime NOT NULL,
   PRIMARY KEY (BibNumber, CheckoutDateTime),
   FOREIGN KEY (ItemType) references ItemType(typeID),
   FOREIGN KEY (ItemCollection) references ItemCollection(collectionID)
);

Each Checkout must have 3 dates (DueDate, CheckinDate, CheckOutDate). Since these are also Books that are being
checked out, we also need the BibNumber, Collection, Type and branchID (and those will be foriegn keys to a Book)
the Primary Key will be the combination of that 4-tuple and the day it was checked out, as well as the library
card since multiple Users can checkout the same Book several times - even in the same day (if the Users are
different) The status as mentioned earlier is just to signify if the book is currently checked out (0), returned
on time (1), or overdue (2)

create table CheckOuts(
  libraryCard INTEGER NOT NULL,
  BibNumber INTEGER NOT NULL,
  checkoutDate datetime NOT NULL,
  checkInDate datetime,
  dueDate datetime NOT NULL,
  branchID INTEGER NOT NULL,
  status INTEGER NOT NULL,
  itemType INTEGER NOT NULL,
  itemCollection INTEGER NOT NULL,
  primary key (libraryCard, BibNumber, checkoutDate, branchID, itemType, itemCollection),
  FOREIGN key (libraryCard) references User(libraryCard),
  FOREIGN key (branchID) references Branches(branchID),
  FOREIGN key (BibNumber, branchID, itemType, itemCollection) references Book(BibNumber, branchID,
ItemCollection, ItemType)
);
```

In addition to these tables, we also have to filter our data:

```
The GoodReads table simply has all the data that we took from the original data set (Rating, Publisher-related
information, ID, Description, pagesCount) as well as the Primary Keys - ISBN and Title.
create table newGoodReads (
   Id TEXT NOT NULL,
   title VARCHAR(500),
   ISBN VARCHAR(20),
   Authors TEXT NOT NULL,
   Rating TEXT NOT NULL,
   PublishYear TEXT,
   PublishMonth TEXT,
   PublishDay TEXT,
   Publisher TEXT,
   RatingDist5 TEXT NOT NULL,
   RatingDist4 TEXT NOT NULL,
   RatingDist3 TEXT NOT NULL,
   RatingDist2 TEXT NOT NULL,
   RatingDist1 TEXT NOT NULL,
   RatingDistTotal TEXT NOT NULL,
   CountsOfReview TEXT NOT NULL,
   Language TEXT NOT NULL,
   pagesNumber TEXT,
   Description TEXT,
   NumOfTextReviews TEXT,
   primary key (ISBN)
);
```

We created an Index on the Primary Key pair as well, to improve the query run times when searching for a rating.

```
CREATE INDEX indexTitle
ON newGoodReads (title, ISBN);
```

Basic loads to load the data from the CSV's

```
LOAD DATA LOCAL INFILE 'C:/Users/russe/Documents/ECE356/Project/books1500003.csv' INTO TABLE GoodReads
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES;

LOAD DATA LOCAL INFILE 'C:/Users/russe/Documents/ECE356/Project/books1500003.csv' INTO TABLE GoodReads
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES;

LOAD DATA LOCAL INFILE 'C:/Users/russe/Documents/ECE356/Project/books1500003.csv' INTO TABLE GoodReads
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES;

LOAD DATA LOCAL INFILE 'C:/Users/russe/Documents/ECE356/Project/seattle.csv' INTO TABLE Book
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\r\n'
```

```sql
IGNORE 1 LINES
(BibNumber, Title, Author, ISBN, PublishYear, Publisher, Subject, ItemType, ItemCollection, @dummy, branchID,
@dummy, ItemCount)
```

Logic to create multiple (BibNumber,ISBN) pairs for each Book that has several ISBNs. Then we will join this with the GoodReads table to filter out all Books in our library that do not have reviews.

```sql
create table preISBNs (
   BibNumber INTEGER NOT NULL,
   ISBN VARCHAR(20) NOT NULL,
   FOREIGN KEY (BibNumber) references Book(BibNumber)
);

insert ignore into preISBNs (BibNumber, ISBN)
select distinct t.BibNumber, replace(j.ISBN, ' ', '')
from book t
join json_table(
   replace(json_array(t.ISBN), ',', '","'),
   '$[*]' columns (ISBN varchar(20) path '$')
) j;
```

This table will represent the “ISBN Exists” relationship that allows us to determine if a Book has a rating in GoodReads.

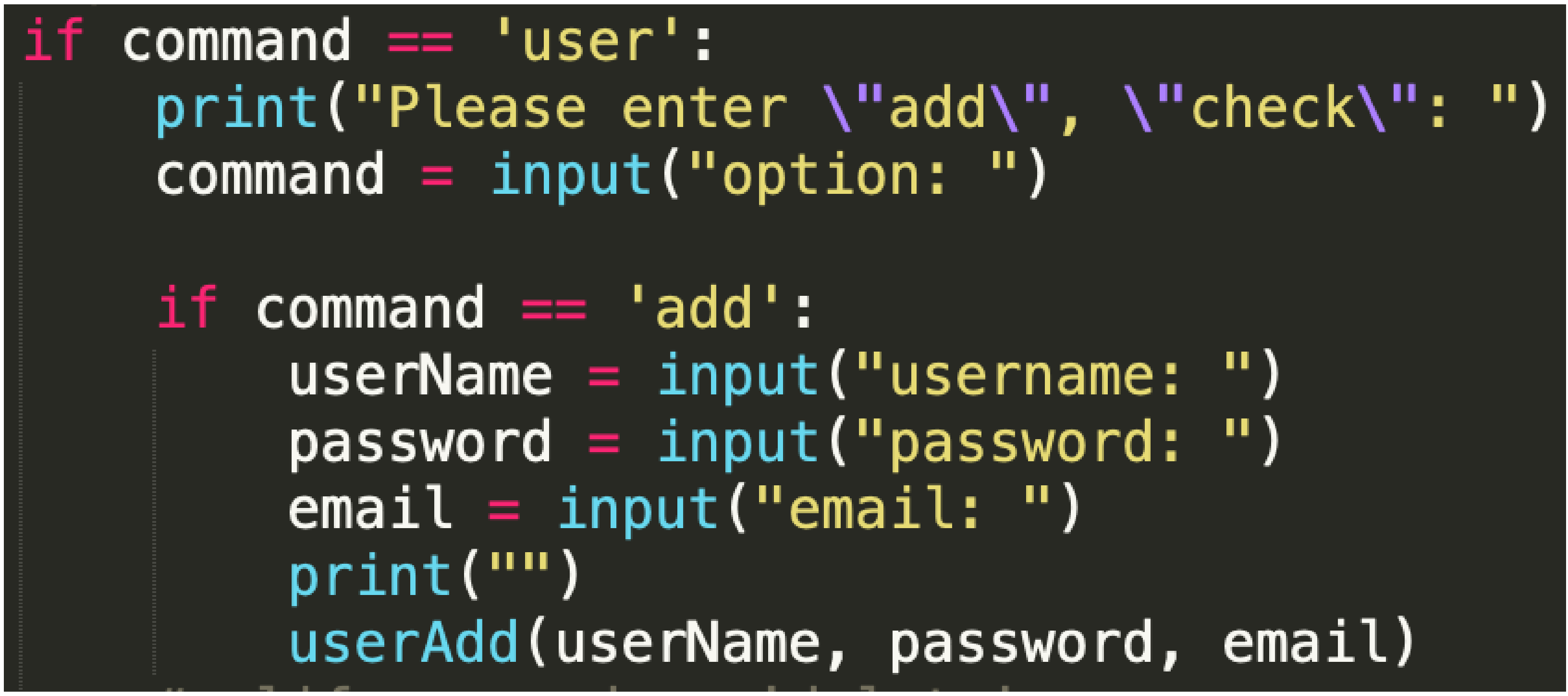
```sql
create table ISBNs (
   BibNumber INTEGER NOT NULL,
   ISBN VARCHAR(20) NOT NULL,
   PRIMARY KEY (BibNumber, ISBN),
   FOREIGN KEY (BibNumber) references Book(BibNumber),
   FOREIGN KEY (ISBN) references GoodReads(ISBN)
);
```

Here we are inner joining the results to filter out any non-existent Seattle Books in GoodReads.

```sql
insert into ISBNs (BibNumber, ISBN) select BibNumber, ISBN from preISBNs inner join goodreads using(isbn);

drop table preISBNs;
```

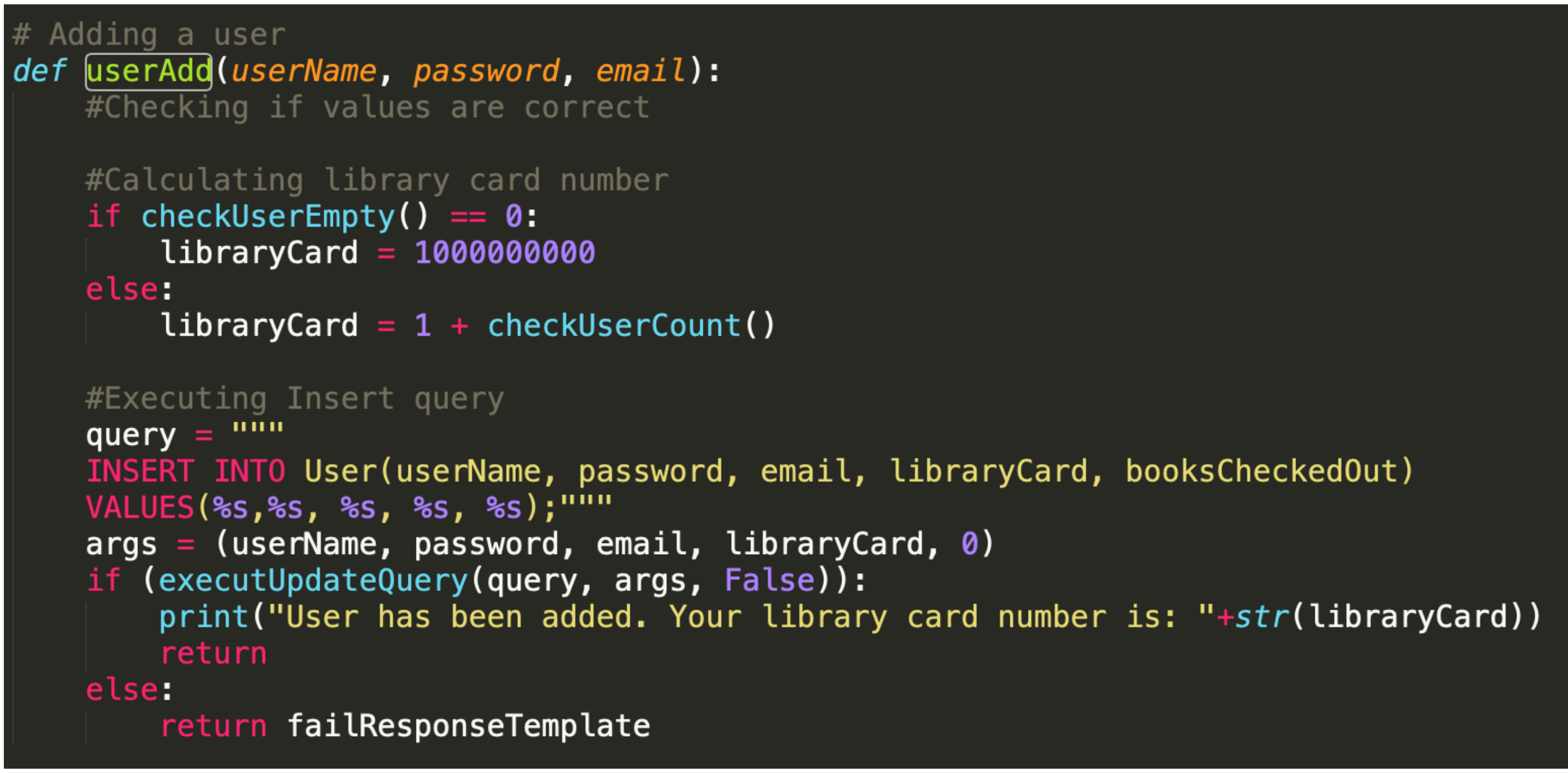
Client Implementation, Sample Functions and Queries

Adding User



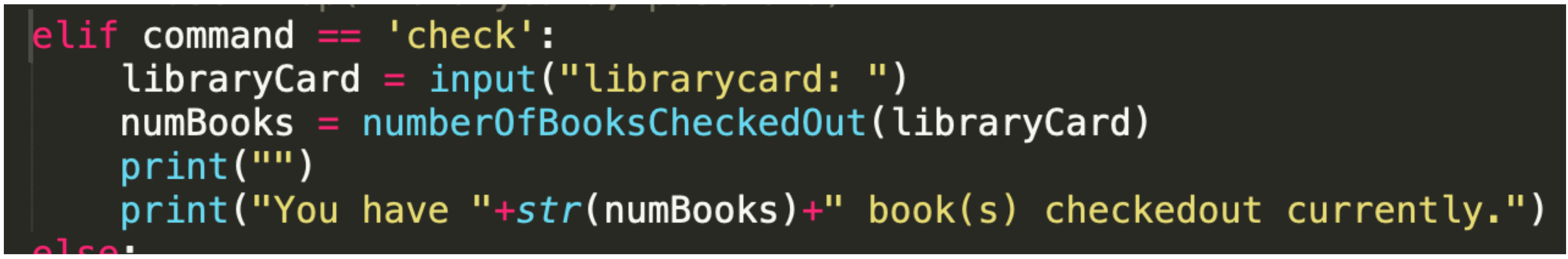
```
if command == 'user':
    print("Please enter \"add\", \"check\": ")
    command = input("option: ")

    if command == 'add':
        userName = input("username: ")
        password = input("password: ")
        email = input("email: ")
        print("")
        userAdd(userName, password, email)
```

If an Admin wants to add a user, we simply ask for the username, password and email of this user. Then, we call the userAdd function. This function simply assigns the user a library card based on the last maximum library card in the database, and executes a simple “Insert Into” query.

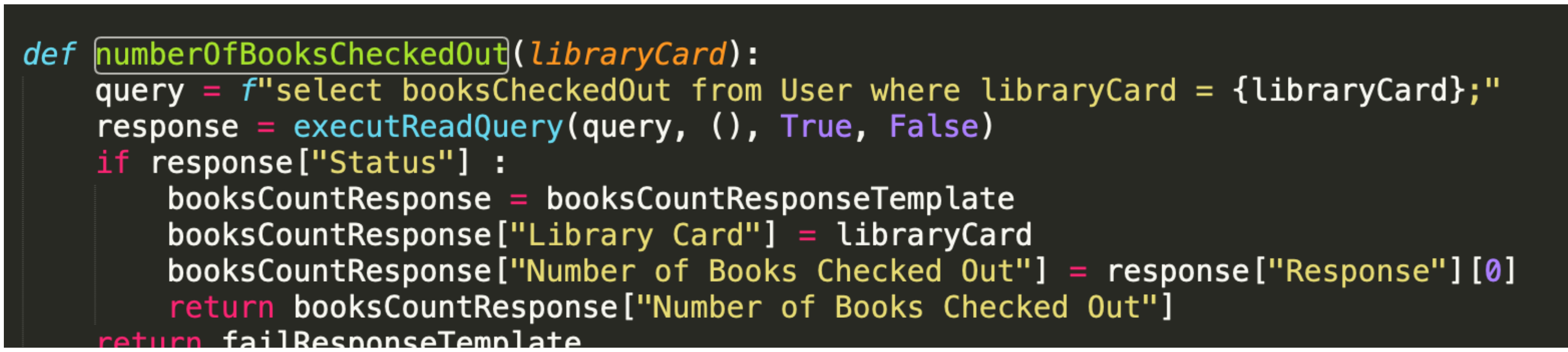


```python
# Adding a user
def userAdd(userName, password, email):
    #Checking if values are correct

    #Calculating library card number
    if checkUserEmpty() == 0:
        libraryCard = 1000000000
    else:
        libraryCard = 1 + checkUserCount()

    #Executing Insert query
    query = """
    INSERT INTO User(userName, password, email, libraryCard, booksCheckedOut)
    VALUES(%s,%s, %s, %s, %s);"""
    args = (userName, password, email, libraryCard, 0)
    if (executUpdateQuery(query, args, False)):
        print("User has been added. Your library card number is: "+str(libraryCard))
        return
    else:
        return failResponseTemplate
```

If an administrator or user wants to instead check the number of checkouts they currently have, we simply call the numberOfUsersCheckedOut function. All this function does is execute a simple “Select From”, where we select the booksCheckedOut attribute.

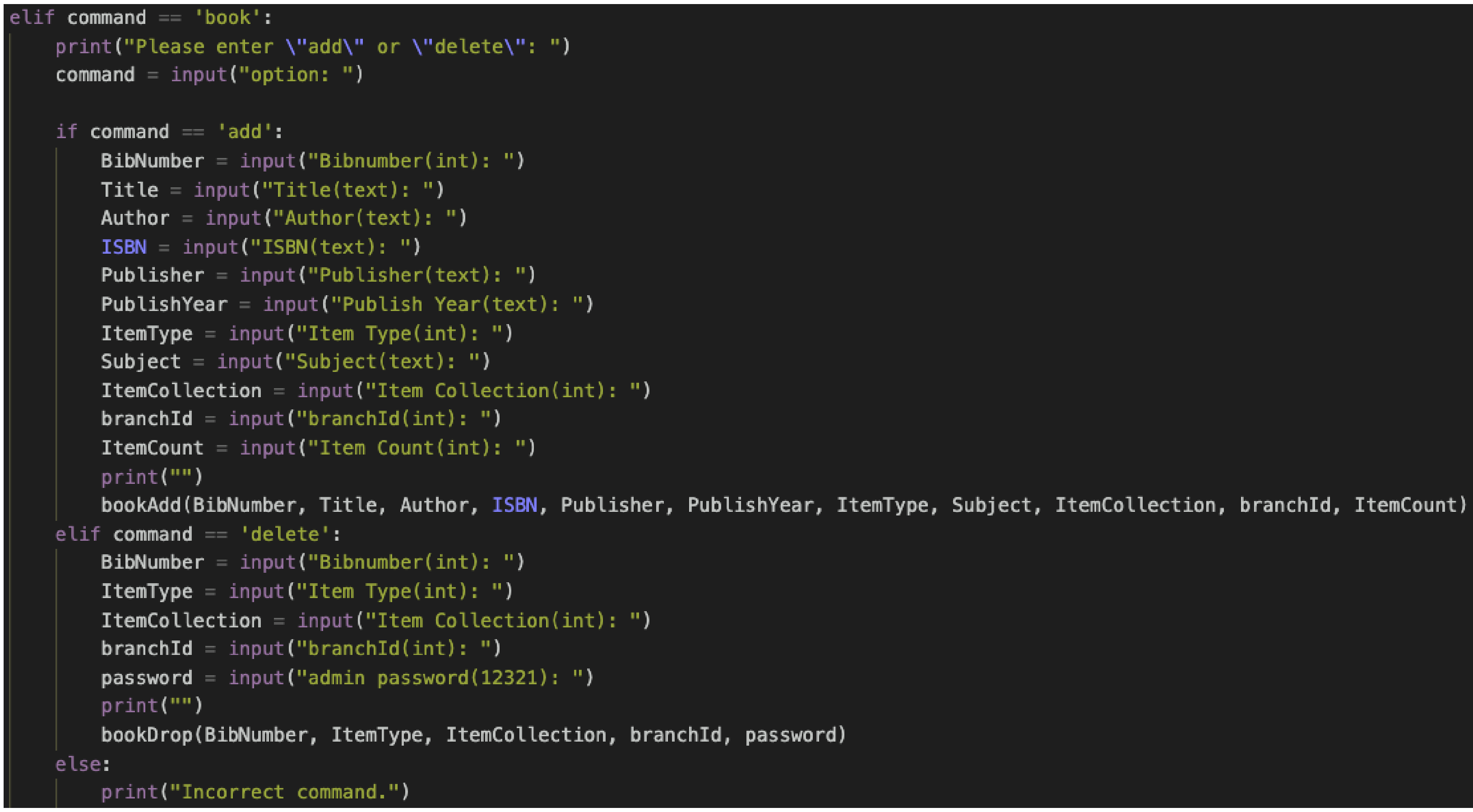


```python
elif command == 'check':
    libraryCard = input("librarycard: ")
    numBooks = numberOfBooksCheckedOut(libraryCard)
    print("")
    print("You have "+str(numBooks)+" book(s) checkedout currently.")
else:
```



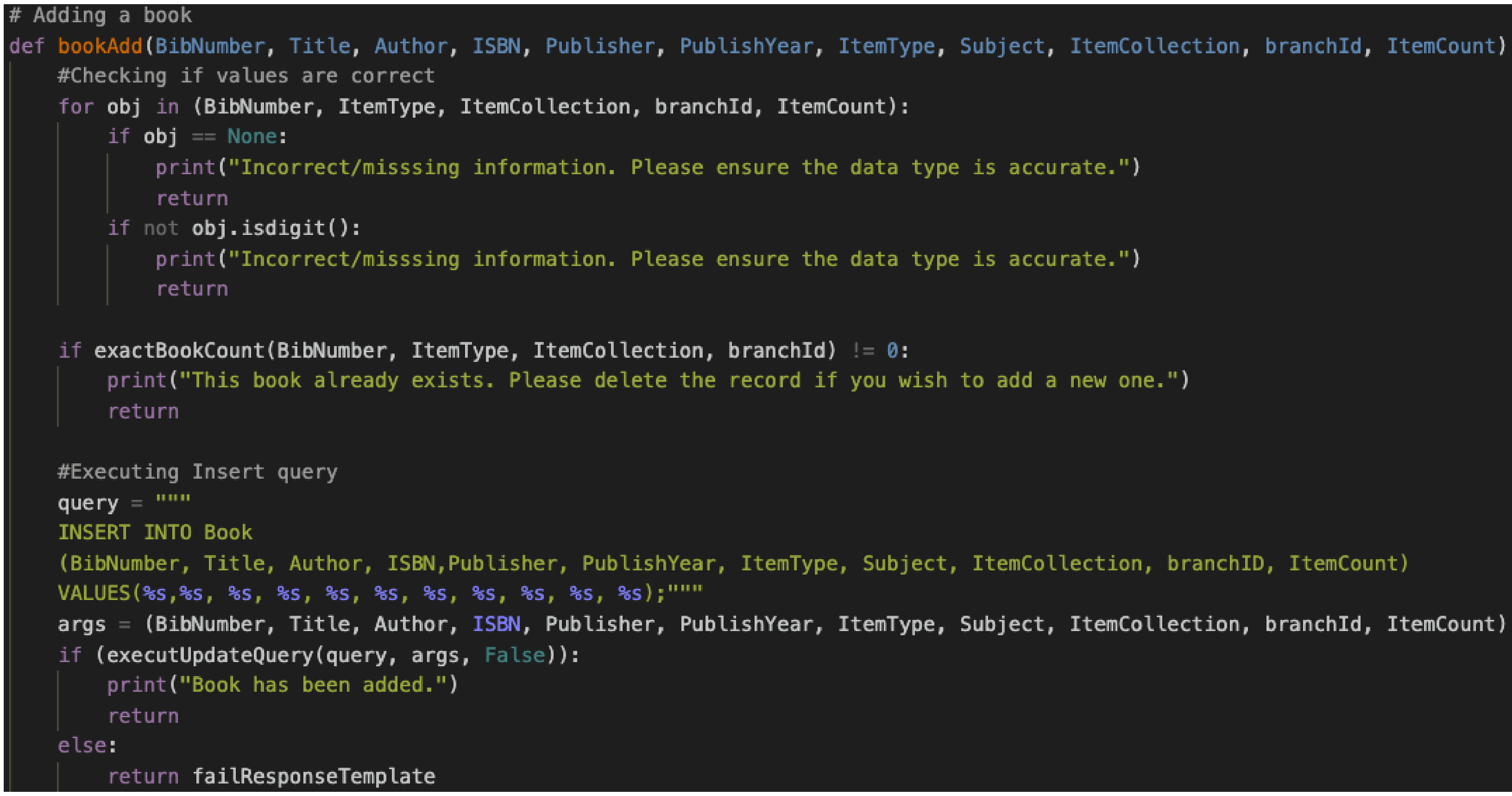
```python
def numberOfBooksCheckedOut(libraryCard):
    query = f"select booksCheckedOut from User where libraryCard = {libraryCard};"
    response = executReadQuery(query, (), True, False)
    if response["Status"] :
        booksCountResponse = booksCountResponseTemplate
        booksCountResponse["Library Card"] = libraryCard
        booksCountResponse["Number of Books Checked Out"] = response["Response"][0]
        return booksCountResponse["Number of Books Checked Out"]
    return failResponseTemplate
```

Adding and Deleting a Book



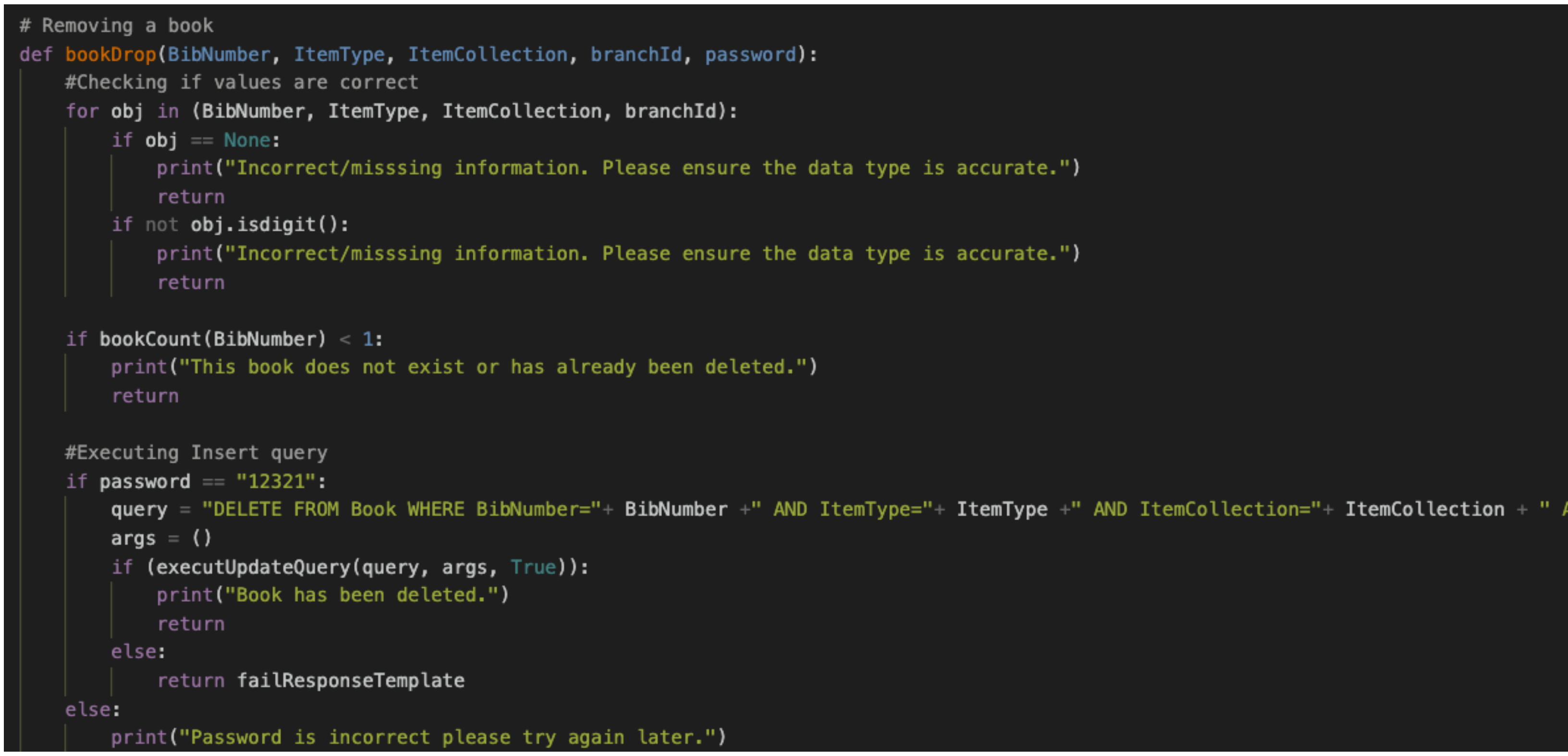
```
elif command == 'book':
    print("Please enter \"add\" or \"delete\": ")
    command = input("option: ")

    if command == 'add':
        BibNumber = input("Bibnumber(int): ")
        Title = input("Title(text): ")
        Author = input("Author(text): ")
        ISBN = input("ISBN(text): ")
        Publisher = input("Publisher(text): ")
        PublishYear = input("Publish Year(text): ")
        ItemType = input("Item Type(int): ")
        Subject = input("Subject(text): ")
        ItemCollection = input("Item Collection(int): ")
        branchId = input("branchId(int): ")
        ItemCount = input("Item Count(int): ")
        print("")
        bookAdd(BibNumber, Title, Author, ISBN, Publisher, PublishYear, ItemType, Subject, ItemCollection, branchId, ItemCount)
    elif command == 'delete':
        BibNumber = input("Bibnumber(int): ")
        ItemType = input("Item Type(int): ")
        ItemCollection = input("Item Collection(int): ")
        branchId = input("branchId(int): ")
        password = input("admin password(12321): ")
        print("")
        bookDrop(BibNumber, ItemType, ItemCollection, branchId, password)
    else:
        print("Incorrect command.")
```

If an Admin wants to add a book, we simply ask for the information of that book such as bib number and title of the book which then we pass into our bookAdd function.



```python
# Adding a book
def bookAdd(BibNumber, Title, Author, ISBN, Publisher, PublishYear, ItemType, Subject, ItemCollection, branchId, ItemCount)
    #Checking if values are correct
    for obj in (BibNumber, ItemType, ItemCollection, branchId, ItemCount):
        if obj == None:
            print("Incorrect/misssing information. Please ensure the data type is accurate.")
            return
        if not obj.isdigit():
            print("Incorrect/misssing information. Please ensure the data type is accurate.")
            return

    if exactBookCount(BibNumber, ItemType, ItemCollection, branchId) != 0:
        print("This book already exists. Please delete the record if you wish to add a new one.")
        return

    #Executing Insert query
    query = """
    INSERT INTO Book
    (BibNumber, Title, Author, ISBN,Publisher, PublishYear, ItemType, Subject, ItemCollection, branchID, ItemCount)
    VALUES(%s,%s, %s, %s, %s, %s, %s, %s, %s, %s, %s);"""
    args = (BibNumber, Title, Author, ISBN, Publisher, PublishYear, ItemType, Subject, ItemCollection, branchId, ItemCount)
    if (executUpdateQuery(query, args, False)):
        print("Book has been added.")
        return
    else:
        return failResponseTemplate
```

In the bookAdd function we check if the arguments are of valid type and create the book insert query. This query along with the argument are then passed to the execute update function where it will be executed and the response is returned to ensure it was successful.

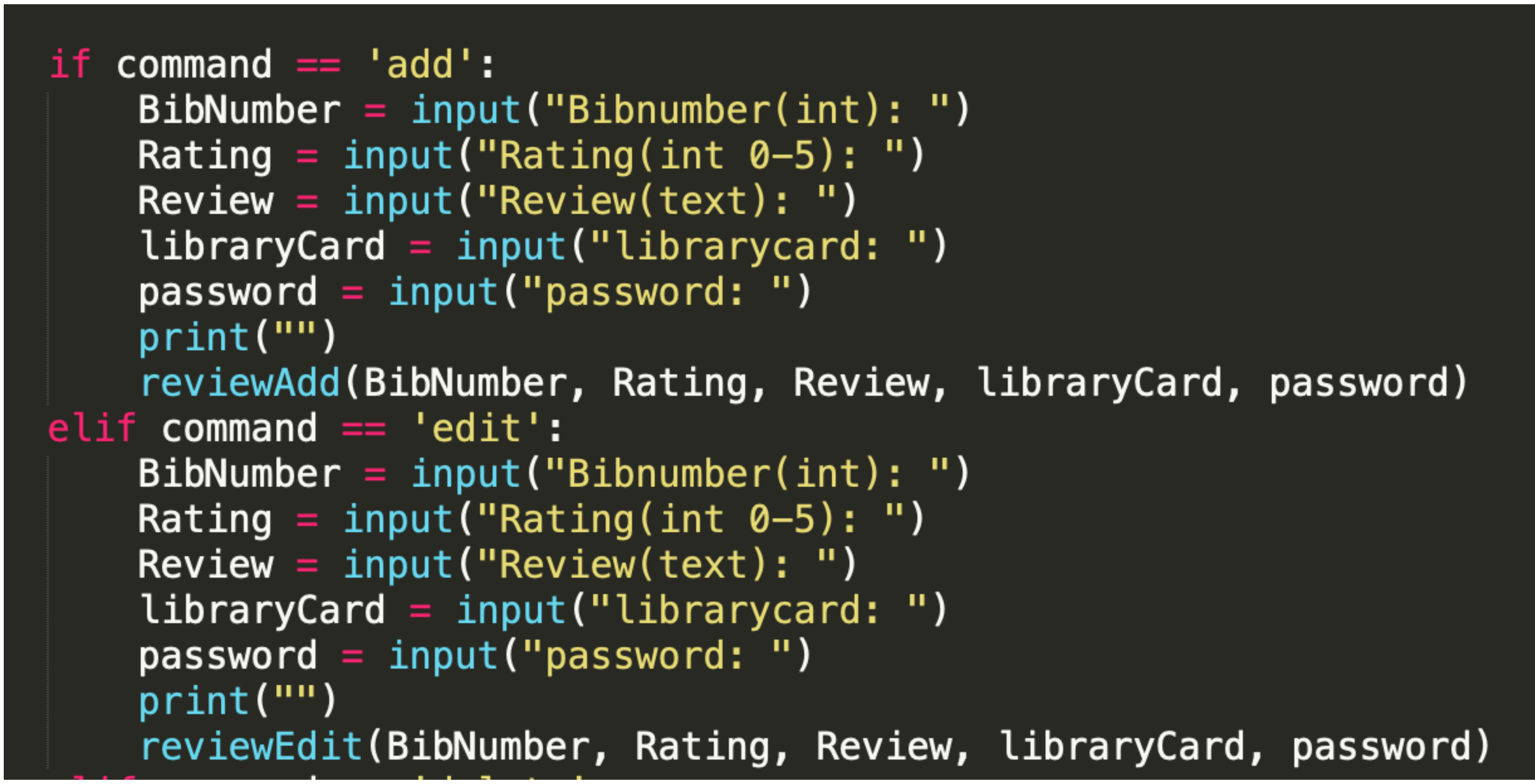


```python
# Removing a book
def bookDrop(BibNumber, ItemType, ItemCollection, branchId, password):
    #Checking if values are correct
    for obj in (BibNumber, ItemType, ItemCollection, branchId):
        if obj == None:
            print("Incorrect/misssing information. Please ensure the data type is accurate.")
            return
        if not obj.isdigit():
            print("Incorrect/misssing information. Please ensure the data type is accurate.")
            return

    if bookCount(BibNumber) < 1:
        print("This book does not exist or has already been deleted.")
        return

    #Executing Insert query
    if password == "12321":
        query = "DELETE FROM Book WHERE BibNumber="+ BibNumber +" AND ItemType="+ ItemType +" AND ItemCollection="+ ItemCollection + " A
        args = ()
        if (executUpdateQuery(query, args, True)):
            print("Book has been deleted.")
            return
        else:
            return failResponseTemplate
    else:
        print("Password is incorrect please try again later.")
```

If the user wishes to delete a book we again ask for the primary key information of the book such as bib number and branch Id along with the admin password to ensure the user has permission to delete books. This then calls our bookDrop function which creates the delete query and ensures the query is successful.

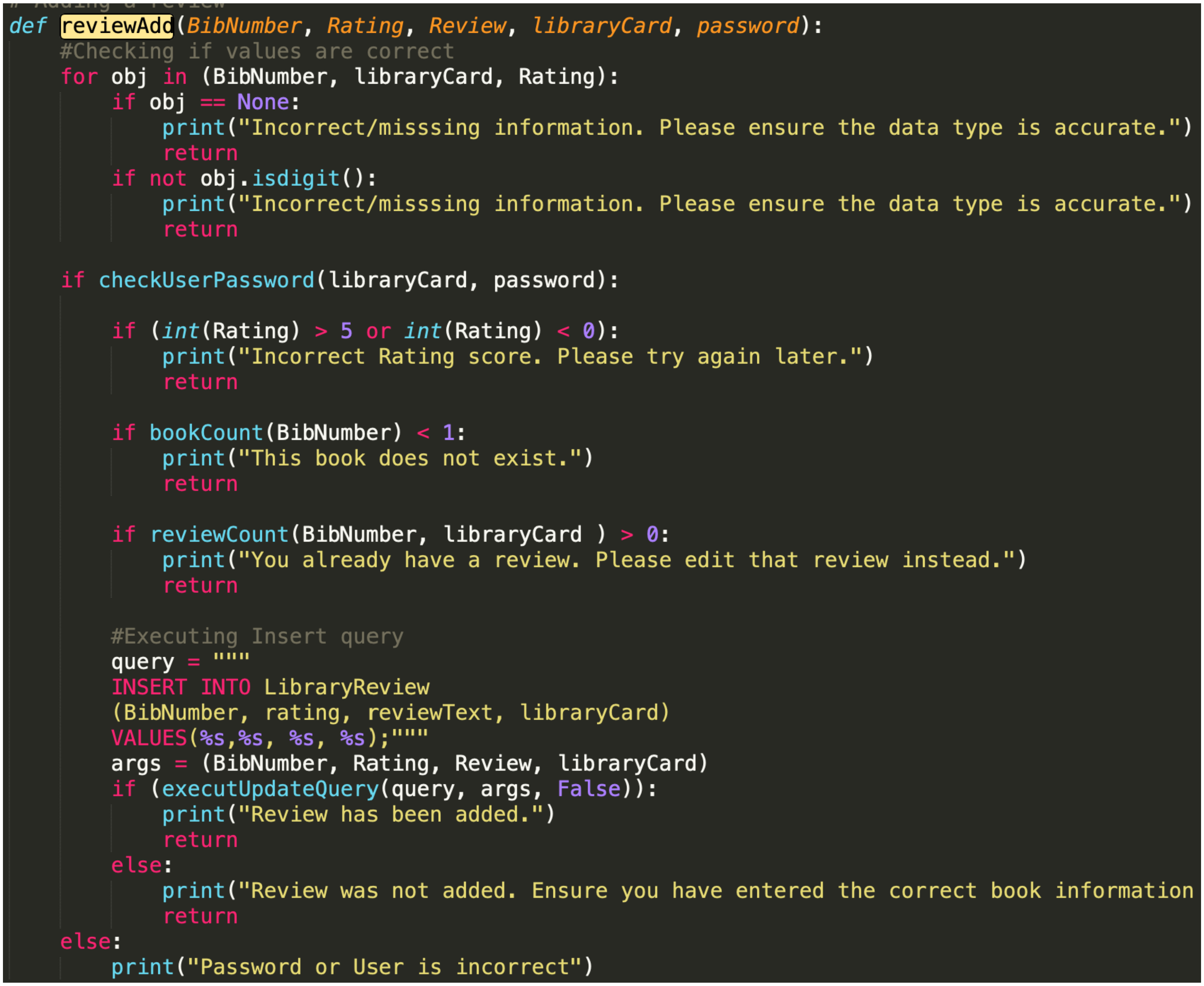
Adding, Editing, Deleting or Viewing a Review

The client has a variety of options when it comes to reviews in the Seattle Library. If a User wants to add a new rating or update an existing one, we simply call reviewAdd and reviewUpdate. Both of these functions work the same way with just a difference in the query (Insert Into vs update table), so we will only show how one of them works.



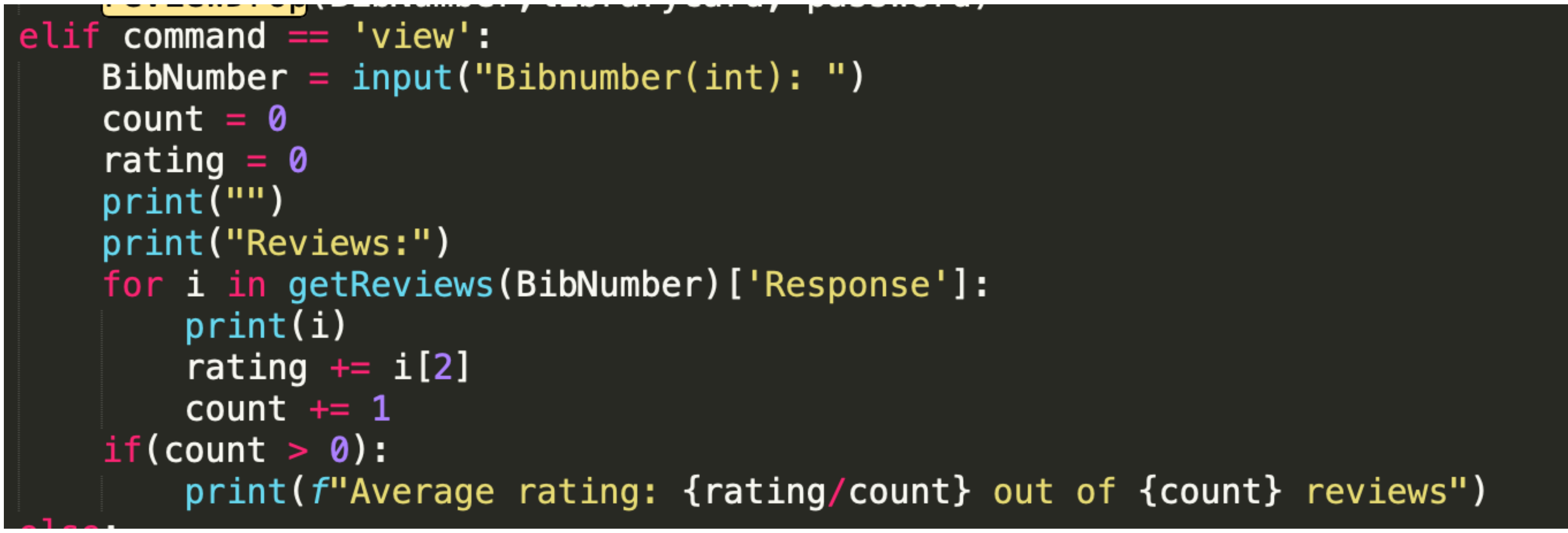
```python
if command == 'add':
    BibNumber = input("Bibnumber(int): ")
    Rating = input("Rating(int 0-5): ")
    Review = input("Review(text): ")
    libraryCard = input("librarycard: ")
    password = input("password: ")
    print("")
    reviewAdd(BibNumber, Rating, Review, libraryCard, password)
elif command == 'edit':
    BibNumber = input("Bibnumber(int): ")
    Rating = input("Rating(int 0-5): ")
    Review = input("Review(text): ")
    libraryCard = input("librarycard: ")
    password = input("password: ")
    print("")
    reviewEdit(BibNumber, Rating, Review, libraryCard, password)
```

You can see in the picture below that the query for adding a book is a simple “Insert Into” query, where we insert this review into the LibraryReview table.



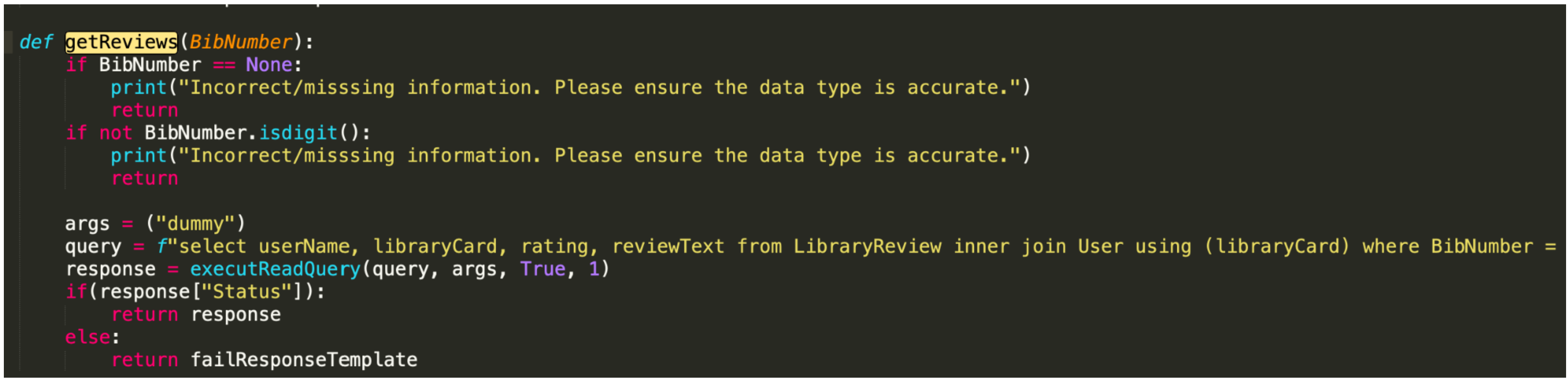
```python
# Adding a review
def reviewAdd(BibNumber, Rating, Review, libraryCard, password):
    #Checking if values are correct
    for obj in (BibNumber, libraryCard, Rating):
        if obj == None:
            print("Incorrect/misssing information. Please ensure the data type is accurate.")
            return
        if not obj.isdigit():
            print("Incorrect/misssing information. Please ensure the data type is accurate.")
            return

    if checkUserPassword(libraryCard, password):

        if (int(Rating) > 5 or int(Rating) < 0):
            print("Incorrect Rating score. Please try again later.")
            return

        if bookCount(BibNumber) < 1:
            print("This book does not exist.")
            return

        if reviewCount(BibNumber, libraryCard ) > 0:
            print("You already have a review. Please edit that review instead.")
            return

        #Executing Insert query
        query = """
        INSERT INTO LibraryReview
        (BibNumber, rating, reviewText, libraryCard)
        VALUES(%s,%s, %s, %s);"""
        args = (BibNumber, Rating, Review, libraryCard)
        if (executUpdateQuery(query, args, False)):
            print("Review has been added.")
            return
        else:
            print("Review was not added. Ensure you have entered the correct book information
            return
    else:
        print("Password or User is incorrect")
```

This is nearly identical to the logic of editing a review. Now, when deleting a review, the BibNumber, LibraryCard and the password of the user are the only thing that are needed, and this time the query is a “delete From” where we remove all the reviews matching this information from the LibraryCheckouts. When we want to view the existing ratings of a book, we only require the BibNumber you want to see the reviews for.



```
elif command == 'view':
    BibNumber = input("Bibnumber(int): ")
    count = 0
    rating = 0
    print("")
    print("Reviews:")
    for i in getReviews(BibNumber)['Response']:
        print(i)
        rating += i[2]
        count += 1
    if(count > 0):
        print(f"Average rating: {rating/count} out of {count} reviews")
```

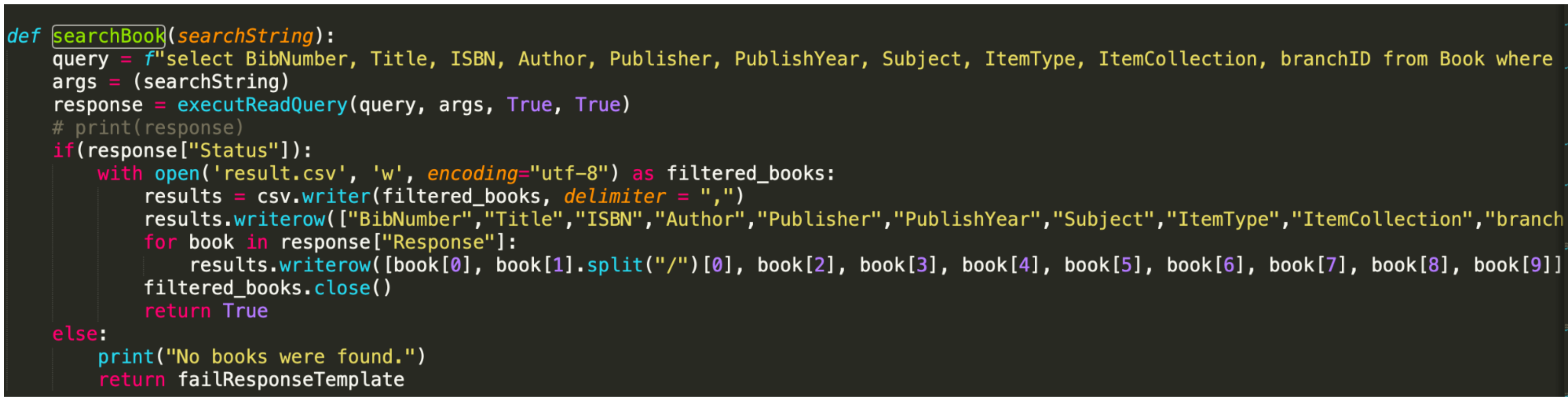
We will output to a user the average rating of this book, as well as all the existing reviews for it = both of which are simple “select from” queries



```
def getReviews(BibNumber):
    if BibNumber == None:
        print("Incorrect/misssing information. Please ensure the data type is accurate.")
        return
    if not BibNumber.isdigit():
        print("Incorrect/misssing information. Please ensure the data type is accurate.")
        return

    args = ("dummy")
    query = f"select userName, libraryCard, rating, reviewText from LibraryReview inner join User using (libraryCard) where BibNumber =
    response = executReadQuery(query, args, True, 1)
    if(response["Status"]):
        return response
    else:
        return failResponseTemplate
```

Searching For/By

Almost all of our search functions work in a similar manner, with small tweaks to allow the user to specify what they want to search for and by. For this reason, we will only discuss one of the functions, and hope that it is easy to understand the way the rest of them work based on this example. Before we begin, it is important to note that oftentimes search results can be extensive - there can be over 1000 books that have type A for example. As a result, we output the majority of the results to a CSV to make it easier for the User/Admin to read and parse if need be. Let’s take a look at a core search functionality: finding a book by a title. In this case, the user selects search → title → and is then prompted to give us the title he wants to search by. We will then execute a simple “select from” but also incorporate the regexp on the search string that we were provided - then print these results to a CSV. Other functions that search by exact keywords (type, collection) will not include regex in their queries and will instead replace it with “select from where equals”

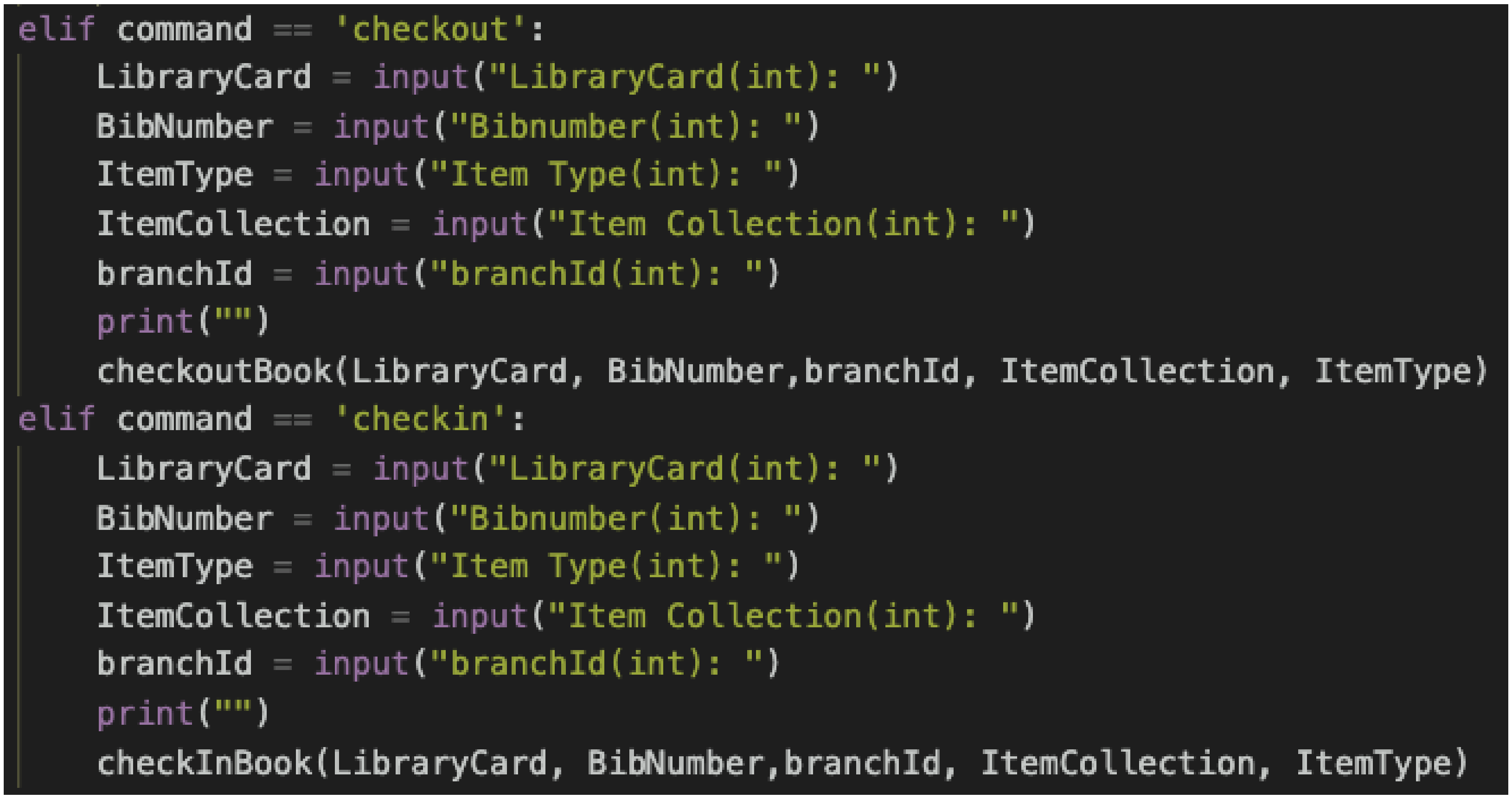


```
def searchBook(searchString):
    query = f"select BibNumber, Title, ISBN, Author, Publisher, PublishYear, Subject, ItemType, ItemCollection, branchID from Book where
    args = (searchString)
    response = executReadQuery(query, args, True, True)
    # print(response)
    if(response["Status"]):
        with open('result.csv', 'w', encoding="utf-8") as filtered_books:
            results = csv.writer(filtered_books, delimiter = ",")
            results.writerow(["BibNumber","Title","ISBN","Author","Publisher","PublishYear","Subject","ItemType","ItemCollection","branch
            for book in response["Response"]:
                results.writerow([book[0], book[1].split("/")[0], book[2], book[3], book[4], book[5], book[6], book[7], book[8], book[9]]
            filtered_books.close()
            return True
    else:
        print("No books were found.")
        return failResponseTemplate
```

The full query used here is
query = f"select BibNumber, Title, ISBN, Author, Publisher, PublishYear, Subject, ItemType, ItemCollection, branchID from Book where Title regexp '{searchString}'"

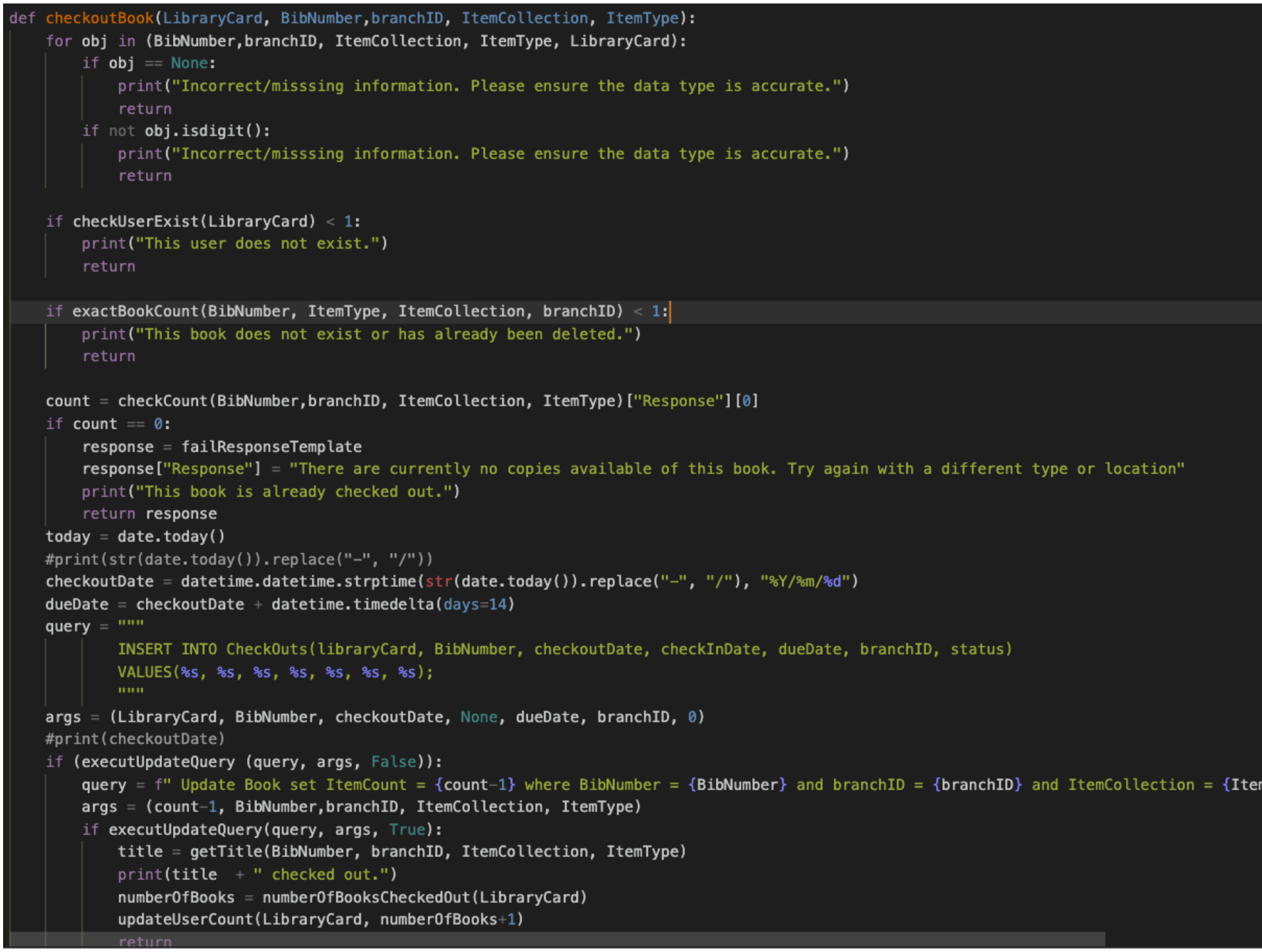
Please keep in mind that this is not all what our client does; these are simply functions that we used to demonstrate how a basic functionality of the client can be implemented using python + sql. For an in depth analysis of all the functionalities, please see the demo video linked at the top.

Checking Out/In a Book
If the user wishes to checkout a book they would simply provide their library card number as well as the book's details to the client which then calls the checkoutBook function.



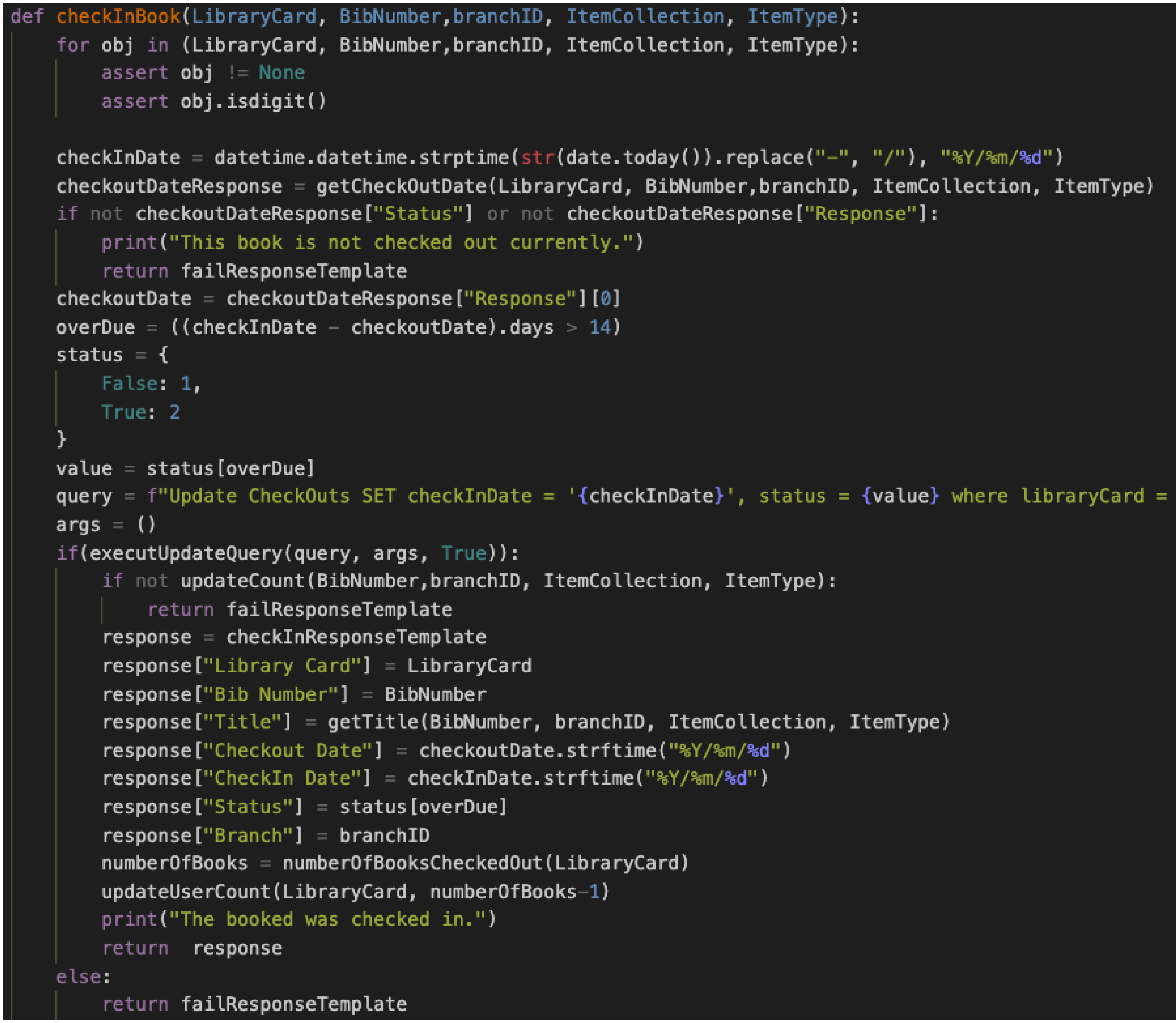
```
elif command == 'checkout':
    LibraryCard = input("LibraryCard(int): ")
    BibNumber = input("Bibnumber(int): ")
    ItemType = input("Item Type(int): ")
    ItemCollection = input("Item Collection(int): ")
    branchId = input("branchId(int): ")
    print("")
    checkoutBook(LibraryCard, BibNumber,branchId, ItemCollection, ItemType)
elif command == 'checkin':
    LibraryCard = input("LibraryCard(int): ")
    BibNumber = input("Bibnumber(int): ")
    ItemType = input("Item Type(int): ")
    ItemCollection = input("Item Collection(int): ")
    branchId = input("branchId(int): ")
    print("")
    checkInBook(LibraryCard, BibNumber,branchId, ItemCollection, ItemType)
```

In the checkout function we ensure the arguments are of the correct type and that both the user exists as well as the book exists. We then check if there are any available copies of the book for the user to check out. If all of these are true, we create the checkout insert query using today's date and add that to the CheckOuts table.



```
def checkoutBook(LibraryCard, BibNumber,branchID, ItemCollection, ItemType):
    for obj in (BibNumber,branchID, ItemCollection, ItemType, LibraryCard):
        if obj == None:
            print("Incorrect/misssing information. Please ensure the data type is accurate.")
            return
        if not obj.isdigit():
            print("Incorrect/misssing information. Please ensure the data type is accurate.")
            return

    if checkUserExist(LibraryCard) < 1:
        print("This user does not exist.")
        return

    if exactBookCount(BibNumber, ItemType, ItemCollection, branchID) < 1:
        print("This book does not exist or has already been deleted.")
        return

    count = checkCount(BibNumber,branchID, ItemCollection, ItemType)["Response"][0]
    if count == 0:
        response = failResponseTemplate
        response["Response"] = "There are currently no copies available of this book. Try again with a different type or location"
        print("This book is already checked out.")
        return response
    today = date.today()
    #print(str(date.today()).replace("-", "/"))
    checkoutDate = datetime.datetime.strptime(str(date.today()).replace("-", "/"), "%Y/%m/%d")
    dueDate = checkoutDate + datetime.timedelta(days=14)
    query = """
            INSERT INTO CheckOuts(libraryCard, BibNumber, checkoutDate, checkInDate, dueDate, branchID, status)
            VALUES(%s, %s, %s, %s, %s, %s, %s);
            """
    args = (LibraryCard, BibNumber, checkoutDate, None, dueDate, branchID, 0)
    #print(checkoutDate)
    if (executUpdateQuery (query, args, False)):
        query = f" Update Book set ItemCount = {count-1} where BibNumber = {BibNumber} and branchID = {branchID} and ItemCollection = {Iter
        args = (count-1, BibNumber,branchID, ItemCollection, ItemType)
        if executUpdateQuery(query, args, True):
            title = getTitle(BibNumber, branchID, ItemCollection, ItemType)
            print(title  + " checked out.")
            numberOfBooks = numberOfBooksCheckedOut(LibraryCard)
            updateUserCount(LibraryCard, numberOfBooks+1)
            return
```

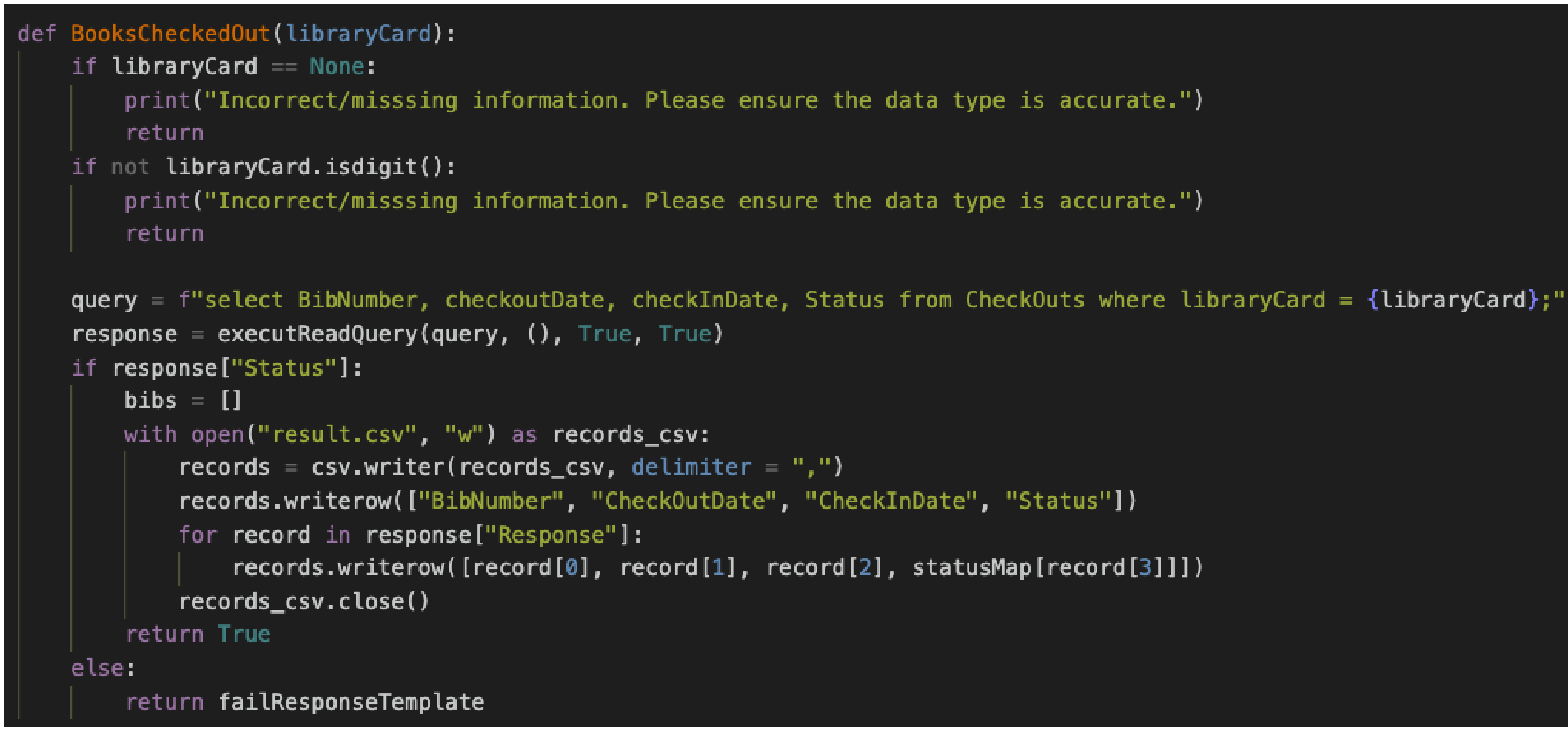
When checking in a book we again ask for the user's library card and the information of the book we are returning. We then check the arguments types and ensure the book is checked out at this moment so that we could return it. We then check if the book is overdue or in the correct return period and if all of this is correct we create the update command to insert a return date into the CheckOuts table entry for that book. We then update the available count for the book and the count of the user's checked out books.



```
def checkInBook(LibraryCard, BibNumber,branchID, ItemCollection, ItemType):
    for obj in (LibraryCard, BibNumber,branchID, ItemCollection, ItemType):
        assert obj != None
        assert obj.isdigit()

    checkInDate = datetime.datetime.strptime(str(date.today()).replace("-", "/"), "%Y/%m/%d")
    checkoutDateResponse = getCheckOutDate(LibraryCard, BibNumber,branchID, ItemCollection, ItemType)
    if not checkoutDateResponse["Status"] or not checkoutDateResponse["Response"]:
        print("This book is not checked out currently.")
        return failResponseTemplate
    checkoutDate = checkoutDateResponse["Response"][0]
    overDue = ((checkInDate - checkoutDate).days > 14)
    status = {
        False: 1,
        True: 2
    }
    value = status[overDue]
    query = f"Update CheckOuts SET checkInDate = '{checkInDate}', status = {value} where libraryCard =
    args = ()
    if(executUpdateQuery(query, args, True)):
        if not updateCount(BibNumber,branchID, ItemCollection, ItemType):
            return failResponseTemplate
        response = checkInResponseTemplate
        response["Library Card"] = LibraryCard
        response["Bib Number"] = BibNumber
        response["Title"] = getTitle(BibNumber, branchID, ItemCollection, ItemType)
        response["Checkout Date"] = checkoutDate.strftime("%Y/%m/%d")
        response["CheckIn Date"] = checkInDate.strftime("%Y/%m/%d")
        response["Status"] = status[overDue]
        response["Branch"] = branchID
        numberOfBooks = numberOfBooksCheckedOut(LibraryCard)
        updateUserCount(LibraryCard, numberOfBooks-1)
        print("The booked was checked in.")
        return  response
    else:
        return failResponseTemplate
```

Viewing Checkouts

If the user wished to view their previous and current checkouts they would simply enter their library card which we then pass to the BooksCheckedOut function.



```python
def BooksCheckedOut(libraryCard):
    if libraryCard == None:
        print("Incorrect/misssing information. Please ensure the data type is accurate.")
        return
    if not libraryCard.isdigit():
        print("Incorrect/misssing information. Please ensure the data type is accurate.")
        return

    query = f"select BibNumber, checkoutDate, checkInDate, Status from CheckOuts where libraryCard = {libraryCard};"
    response = executReadQuery(query, (), True, True)
    if response["Status"]:
        bibs = []
        with open("result.csv", "w") as records_csv:
            records = csv.writer(records_csv, delimiter = ",")
            records.writerow(["BibNumber", "CheckOutDate", "CheckInDate", "Status"])
            for record in response["Response"]:
                records.writerow([record[0], record[1], record[2], statusMap[record[3]]])
            records_csv.close()
        return True
    else:
        return failResponseTemplate
```

In this function we check if the library card is of correct type and then create a select query using the library card as a where condition to ensure we select the correct user information. We then loop through the response and generate a result.csv with the information.

Testing Flow

We want to preface this by mentioning that we had significantly more intensive testing in our client; ranging from incorrect commands, to type checking and query errors. However, for the sake of simplicity we have just decided to provide you the barebones approach to get the main functionality of this client. We didn't provide exact test data because it may be hard for you to know what each integer/key/argument represents; so we opted to simply explain the flow instead.

Create User

Creating a user that does not already exist in the database:
Input "user" → input "add" → (username, password, userName) → `"User has been added. Your library card number is: "+str(libraryCard)"`
Creating a user that already exists in the database:
`{"Error" : "Unable to execute query","Status": False}`

Get Number Of Existing Checkouts:

Input("user") → input ("check") → (libraryCard) → `"You have "+str(numBooks)+" book(s) checked out currently."`
If the libraryCard is invalid:
`{"Error" : "Unable to execute query","Status": False}`

Add/Remove Book:

Input "book" → "add" → (BibNumber, Title, Author, ISBN, Publisher, PublishYear, ItemType, Subject, ItemCollection, branchID, ItemCount) → `"Book has been added"`
If book with same BibNumber, ItemType, ItemCollection, branchID already exists:
`"This book already exists. Please delete the record if you wish to add a new one."`
If args are incorrect:
`"Incorrect/missing information. Please ensure the data type is accurate."`
Else:
`{"Error" : "Unable to execute query","Status": False}`

Input "book" → "remove" → (BibNumber, Title, Author, ISBN, Publisher, PublishYear, ItemType, Subject, ItemCollection, branchID, ItemCount) → `"Book has been deleted"`
If password incorrect:
`"Password is incorrect, please try again later."`
If no book with BibNumber, ItemType, ItemCollection, branchID exists:
`"This book does not exist or has already been deleted"`
If args are incorrect:
`"Incorrect/missing information. Please ensure the data type is accurate."`
Else:
`{"Error" : "Unable to execute query","Status": False}`

Checkout Book:

Input “checkout → (libraryCard, bibNumber, itemType, itemCollection, branchID) → `“{book title} checked out”`
Checking out an item with bookCount == 0:
`“This book is already checked out”`
Else:
`{"Error" : "Unable to execute query","Status": False}`

Check in Book
Checking in a book that is already checked out:
Input “checkin” → (libraryCard, bibNumber, itemType, itemCollection, branchID) → `"The results are in the result.csv file."`
Checking in a book that is not checked out or has been checked out by the same user in the last 24 hours:
Input “checkin” → (libraryCard, bibNumber, itemType, itemCollection, branchID) → `"The book was checked in."`
Checking in a book that was not checked out by the user →
Input “checkin” → (libraryCard, bibNumber, itemType, itemCollection, branchID) → `{"Error" : "Unable to execute query","Status": False}`

Add/Edit a Review of a Book:
Adding a review that does not already exist, or editing a review that already exists for this user:
Input (“review”) →Input (“add” or “edit”) → (BibNumber, Rating, ReviewText, LibraryCard, Password) → `“Review has been added/updated”`
If the book does not exist:
`“This book does not exist”`
If the password is incorrect:
`"Password or User is incorrect"`
If the review does not exist:
`"You have no review to edit."`
If query fails:
`"Review was not added/updated. Ensure you have entered the correct book information or do not already have a review for this book."`

Removing review:
Input (“review”) →Input (“delete”) → (BibNumber, LibraryCard, Password) → `“Review has been deleted”`
If query fails:
`"Review was not removed. Ensure you have entered the correct book information or that you have a review for this book."`
If the password is incorrect:
`"Password or User is incorrect"`
If the book does not exist:
`“This book does not exist”`
If the review does not exist:
`"You have no review to delete."`

View Review

Input (“review”) →Input (“view”) → (BibNumber) →

```
“Reviews:
{BibNumber, Rating, ReviewText}
Average rating: {rating/count} out of {count} reviews"
```

If the book does not exist or the query fails:

`{"Error" : "Unable to execute query","Status": False}`

Search for Items:

Input (“search”) → Input(“title”) → (searchString)→`"The results are in the result.csv file."`

Else:

`{"Error" : "Unable to execute query","Status": False}`

Get BibNumber from result.csv

Input(“search”) → Input(“goodreads”) → (BibNumber) → `“{book information}”`

If BibNumber does not exist in ISBNs table (no ISBN matching to goodreads):

`{"Error" : "Unable to execute query","Status": False}`

Else:

`{"Error" : "Unable to execute query","Status": False}`

Check csv to verify results

Input (“search”) → Input(“bibnumber”) → (BibNumber)→`"The results are in the result.csv file."`

Else:

`{"Error" : "Unable to execute query","Status": False}`

Check csv to verify results

Input(“search”) → Input(“category”) → Input(“2”) → “{itemType, itemCollection}”→`"The results are in the result.csv file."`

Else:

`{"Error" : "Unable to execute query","Status": False}`

Check csv to verify results

Input (“search”) → Input(“records”) → (BibNumber)→`"The results are in the records.csv file."`

Else:

`{"Error" : "Unable to execute query","Status": False}`

Check csv to verify results

User Checkout Records :

Input(“Checkouts”) → (libraryCard) → `“The results are in the result.csv file.”`

If the libraryCard is invalid:

`{"Error" : "Unable to execute query","Status": False}`