Project Set: Lecture 4

Matthew Ellison
SPISE

July 23, 2019

1 Library Database

In this mini project you'll build a basic library database system—such as a librarian might use to manage their collection and check books in and out. The program will present the user (the librarian) with a text interface which they can use to add books to the collection, check a book in or out, or check the status of a book. Here's a sample interaction with such a system:

```
Welcome to the Library Database!

Please Enter a Command: ???
Unknown command. Try 'add book', 'check in', 'check out',
'get book info', or 'print catalog'.

Please Enter a Command: print catalog
Library Empty. Add books!

Please Enter a Command: add book
Enter Title: Green Eggs and Ham
Enter Author: Dr Seuss

Please Enter a Command: print catalog

Green Eggs and Ham
Dr Seuss
in library

Please Enter a Command: check out
Enter title: Green Eggs and Ham
```

```
Please Enter a Command: get book info
Enter title: Green Eggs and Ham

Green Eggs and Ham
Dr Seuss
checked out

Please Enter a Command:
```

If you want to go ahead and build a system like this yourself, go for it! The text below walks through the construction of such a system step by step, if you want guidance. If you decide to follow the outline below, it's a good idea to check that everything's working correctly between each step (for example, after finishing Step 1, check that you can create a book object successfully). Even if you feel like building the system yourself it would be a good idea to look at the outline below, after you're done, to see a possibly different approach.

1.1 Step 1: Book Class

Begin by creating a Book `class`. Give it attributes `self.title`, `self.author`, and `self.available`. Title and author can be provided on initialization. Set `self.available` to `'in library'` on initialization.

Next we'll give the Book class a method `print_info` to print its information. You can print the book's information any way you'd like—such as printing the title, then the author, and then the availability.

1.2 Step 2: Library Class

Next we'll create a Library `class`. This will store the Books and provide methods to add books, and check them in and out.

Begin by writing the initialization for the class. It makes sense to create an attribute `self.books`, which will initially be an empty list (since the library doesn't start with any books, at least in the basic version we're building).

Next write a method `add_book`. This should take in a title and author as parameters, create a book from them, and add that book to `self.books`.

Now write a method `get_book_by_title`, which should take a title as input parameter. This should go through the books in `self.books` and, if one book has the given title, return that book object. Otherwise, the method should return `None` or `'no book'` or something like that.

Next write a method `print_info_by_title`. This method should take a title as input parameter, and then use `get_book_by_title` to get the book. If there isn't such a book, have the method print something like `'No book of this title in library'`. Otherwise, have the book print its information by the Book `print_info` method.

Next we'll do write the methods to do book check-in and check-out.

Write a method `check_in_book`, which takes in a title as parameter. Then, like `print_info_by_title`, use `get_book_by_title` to get the book. If there isn't such a book, have the method print something like `'No book of this title in library'`. Otherwise, set the book's available attribute to `'available'`

Define a `check_out_book` method similarly.

Finally, we'll define a method `print_catalog` to print the information for all books in the library. This should run through all the books in the library, calling the Book method `print_info` for each.

1.3 Step 3: User Interface

Now that we have the classes all set up, it's easy to tie things together with a user interface—let's get started. Begin by creating an instance of the Library `class`, which will serve as our library database throughout. Next, do the following in a big `while` loop:

Prompt the user to enter a command. Then use a big `if`/`elif`/`elif`/.../`else` structure to do different things depending on the entered command. For example, if `command == 'add book'`, prompt the user to enter a title and author, and then use the Library `add_book` method to add the book to the library. Else, if the command is `'check out'`, for example, prompt the user to enter a title and then use the Library `check_out_book` method to check the book out. Implement the other features similarly.

The program will go through the while loop forever, carrying out the given commands to run the library system. At this point you have a working program!

1.4 Optional Extensions

1. Improve the interaction with the user. If the user enters an unknown command, tell the user the command is unknown and give suggestions. If the user tries to check out a book which is already checked out, or something like that, notify them that the book is already checked out. Make sure all your input commands end in a space.

2. Make the library persist between runs of the program (right now, if you start the program again, all the books go away). The best way to do this is probably to create `load_library` and `save_library` methods to the Library `class`.

   The save library method would write the library information to a text file, and the `load_library` method would read through the text file and use the information to add all the books. The easiest way to make books persist between runs is to use something we haven't learned—the `pickle` library. If you're interested, look it up and use that to implement `load_library` and `save_library`.

   You'd run `load_library` in the beginning of the program (before the `while` loop) to get all the books, and then you could run `save_library` whenever the library is modified by a method such as `check_out_book` or `add_book`.

3. Libraries also keep track of who's checked out books. Libraries call the people who check out books 'patrons'. Write a patron class (possible attributes: name, phone number, patron number,...) and modify the book and library classes to keep track of who has what books checked out.

2 Blackjack (Optional)

For this mini-project you'll create a simplified blackjack game. We'll be using a simplified version of blackjack which is just the player against the dealer and which ignores some of the game's special rules (like doubling down, splitting, insurance...).

Here's how each round works, if you happen to not have played before:

1. The dealer and player are each dealt two cards. The player can see both of their cards, but only one of the dealer's.
2. The goal is for the player's cards to add to as close to 21 as possible, without going over, and they are repeatedly offered to take additional cards from the deck until they 'bust' (go over 21, this is an immediate loss) or decide to stop.
   (a) Here's how much each card is worth: 2, 3, 4, .., 10 are worth their number; Jack, Queen and King are worth 10; and Ace is worth 1 or 11, whichever gives the player a better hand.
3. If the player doesn't bust, the dealer then flips over their cards. While the dealer has a hand of 16 or less, the dealer will continue to draw more cards. If they bust, the player wins. Otherwise, the player and dealer will compare hands. If the player is closer to 21 they win, otherwise they lose (the dealer wins on a tie).

You've been provided with a library (written by Daniel Ellison) which provides two classes: `Card` (which represents a playing card), and `Deck` (which represents a deck of cards).

Please read the documentation in the beginning of the `cards.py` file to familiarize yourself with these two classes and their usage. Here are some of the basic features (the library supports many more features, such as combining multiple decks and performing specialized shuffles):

1. Calling `Deck()` creates a new deck of cards (standard 52-card deck, no jokers), in unshuffled order.
2. The `shuffle()` method may be used to randomize the order of cards in a `Deck`.
3. The `deal_card()` method will take off the top card of the deck and return a `Card` object.
   (a) You can access a `Card`'s suit with the `get_suit` method (either `'Spades'`, `'Hearts'`, `'Diamonds'`, `'Clubs'`)

   (b) You can access a `Card`'s value with the `get_value` method (either `'A'`, `'2'`, `'3'`, `'4'`, ..., `'J'`, `'Q'`, or `'K'`)

4. Once a card is dealt, it is taken out of the deck, but if the deck is empty and `deal_card()` is called, the deck adds all the cards back in and shuffles. This means you can just keep dealing and don't need to worry about the deck running out (if you'd like to add in the discards manually and shuffle after each round, read the documentation)

Write a blackjack game which is played by user input. If you haven't completed the library catalog system, it might help to complete that first.

Hint: *A good place to get started is to write the function, which given a hand of cards, returns the value of the hand. This is a little tricky because of the way aces are counted (see above), but once you have this function written, the rest should just be getting the details taken care of.*

2.1 Optional Extensions

1. Blackjack is played for money at casinos and the player bets a certain amount each round. This extension is to add in betting to the game. Here's a simplified version (you can go to Blackjack's wikipedia page if you want something more typical of an actual casino):

   (a) When the game starts, provide the user/player with some amount of money.

   (b) Before each round, prompt the player to enter a bet. This happens before they even see any cards.

   (c) If they player wins, the win the amount of their bet. And if they lose, they lose the amount of their bet.

2. Sometimes blackjack is played with two extra rules: '5 Card Charlie, and '6 Card Sam'. These rules say that if the player has a 5 or 6 card hand (without busting) they automatically win. Incorporate this into the game if you'd like.

3. It's generally true that high cards favor the player and are bad for the dealer. For this reason, some players ('card counters') keep track of which cards have been played to get an edge. Casino's don't like card counters because they sometimes make money. That's why casinos shuffle the cards often in blackjack, so the counters can't get much information. Here's a basic technique for counting cards:

   (a) Begin with a count of 0.

   (b) Whenever you see a card from 2-6 you add 1 to the count. If you see a 10, J, Q, K, or A, subtract 1 from the count. 7, 8, and 9 don't effect the count.

   The higher the count, the more you should bet (of course, the count should be reset whenever the deck is shuffled). Write a program to keep track of the count and provide this secret information to the player each round before betting.