Classes

Pieces of this lab are adapted from A Primer on Classes in Python by Aquiles Carattino.

Repeat Warning

In order to be successful in computer science, an important skill that you must develop is understanding how to implement and develop a specification, and knowing when you can take liberty with the specification. You also need to distinguish between implementing and developing a specification. In the course work for CSC111, we ask for some things very precisely. It is important that you read every word of lab and assignment descriptions. Misreading or ignoring directions will result in incorrect software/code.

In this lab, you'll practice working with **classes and object-oriented programming techniques** through several exercises, and finally by modifying a YouTube Playlist program.

Step 1: Getting started with `OOP`

Object-oriented programming (OOP) is a programming paradigm that enables us to specify the operations that can be performed on objects.

Objects and `classes`

When talking about OOP, its hard not to interchange the words `class` and `object`. In truth, the difference between them is quite subtle: an `object` is an *instance* (or "a specific example") of a `class`. For example, `"hello!"` is an *instance* of a string `str`, `3` is an *instance* of an integer `int`, and `54.12` is an *instance* of a decimal number `float`.

While we are learning, we will try to be precise with our language: we will use the word `class` when referring to the *type of variable*, while we will use the word `object` to the *specific variable itself*. This will become clearer as we progress.

Defining a `class`

Let's start by defining a class. A class definition consists of the class name, and a **constructor** `__init__`, where attributes are typically given values.

```
class Person():
    def __init__(self):
        self.name = ""
```

Here our class `Person` has one attribute `name` and no methods.

Create a new file/document in Thonny and save it locally with the file name `lab13p1.py`. Add the course header with the relevant information.

Initializing a class

Below is a more complete example (add it to your lab file):

```
class Person():
    def __init__(self, name):
      self.name = name

    def sayHi(self):
      print("Hi, my name is", self.name)

def main():
  my_friend = Person("Bob")
  my_friend.sayHi()
if __name__ == "__main__":
    main()
```

Instantiating is the moment in which we call the constructor and assign it's value to a variable. The instantiation of the class happens when we say `my_friend = Person(`Bob`)`. *Notice that this instantiation is outside the class.*

In the first example, the constructor did not take `name` as a parameter. This would cause issues if someone calls the `sayHi()` method before actually having a name stored. To enforce all people to have names, we add a required parameter to the constructor.

To really drive home what's happening here, let's create three different instances, and then call the `.sayHi()` method on each of them:

```
my_friend_1 = Person("Alice")
my_friend_2 = Person("Bob")
my_friend_3 = Person("Chloe")
my_friend_1.sayHi()
my_friend_2.sayHi()
my_friend_3.sayHi()
```

Even though the `.sayHi()` method was only defined in one place (the definition of the `Person class`), it has slightly different behavior on each `instance`. That's because each `Person` has their own `name` attribute, and **that** is what the `.sayHi()` method is referencing when it gets called on each `instance`.

Another cool thing about attributes? They can be **any** data type - even other classes!

- Add the following attributes to your `Person` class:
    - `age` (initialized to a random integer between 1 and 99)
    - `favoriteColor` (initialized by passing an additional parameter to the constructor)
- Modify the `.sayHi()` method to include these attributes in the printed message, e.g. `Hi, my name is Kacey, I am 21 years old and my favorite color is green.`

There is also an important detail that was omitted this far, the presence of `self` in the declaration of the method. All the methods in python take a first input variable called self, referring to the class itself. For the time being do not stress yourself about it, but bear in mind that when you define a new method, you should always include the `self`, but when calling the method you should never include it. You can also write methods that do not take any input, but still will have the `self` in them, as in the method `.sayHi()`.

In an MSWord document. Answer the following short answer questions:

- SA Question #1: What other attributes would you add to the person class?
- SA Question #2: What other methods would you add to the person class?
- SA Question #3: Suppose you wanted to confirm the age of the person, write a `.getAge()` method that returns the age of the person.
- SA Question #4: Suppose you wanted to check whether or not the person was a minor (age less than 21 years old). Write the method `.isMinor()` that returns true if the age of the person is less than 21, and false otherwise.

Step 2: A YouTube Playlist

Create a new file/document in Thonny and save it locally with the file name `lab13p2.py`. Add the course header with the relevant information.

Now that we have started to get the hang of classes, let's create a YouTube Playlist. Below is a simple program to play songs you select from YouTube. **Try the program in Thonny, BUT do not copy this code into your lab file. We are going to build the example from scratch.**

```
# A cool tool to manage a simple playlist
# TRY IT!

import webbrowser
from time import sleep


def main():
  # Initialize empty playlist
  playlist = []
  keepGoing = ""
  while(keepGoing != "DONE"):

    # Initialize empty song dictionary
    song = {}

    # Ask for info about song
    song["title"] = input("Song title: ")
    song["url"] = input("YouTube video: ")
    song["duration"] = input("Video duration (MM:SS): ")

    playlist.append(song)

    keepGoing = input("Add another song? YES to continue, DONE to end: ")

  # Print out the playlist
  num = 0

  for song in playlist:
    print(num,song["title"], "(" + song["duration"] + ")")
```

```
    num+=1

  # choose a song to play
  playme = eval(input("Enter the number of the song to play (-1 to quit): "))

  while playme != -1:
    #display the selection and begin the play
    print("PLAYING: " , playlist[playme])
    print("FROM: " , playlist[playme]["url"])
    webbrowser.open(playlist[playme]["url"])
    #get the time to wait
    myduration = playlist[playme]["duration"]
    time = myduration.split(":")
    total_seconds = int(time[0])*60 + int(time[1])
    sleep(total_seconds) # Wait until the video is over
    #report song end
    print("Song is over!")
    #ask if the user would like to play another song
    playme = eval(input("Enter the number of the song to play (-1 to quit): "))



if __name__ == "__main__":
  main()


# EXTRA FOR EXPERTS:
# Can you save the playlist and load it back when you run the program again?
#
```

Notice how our playlist is actually just a list of dictionaries? This is not terrible, but it means that our code for printing information about the songs (as well as playing them) is completely separated from where the relevant data is stored. Now that we know how to define our own `class`, let's see if we can do better.

Song Class

We will begin by defining a `class` to store the songs. Write the song class given the following specification (i.e., what needs to go in it). Be sure to comment your code as you go.

Class **Song**: A class that represents a single song.

Attributes:

- title: Song title as a string
- url: Song duration as a string
- duration: Song url as a string

Contructor and Methods:

- constructor (title, url, duration): Takes three strings and saves them as attributes.
- printInfo ( ): Prints the information about the song in a friendly way (e.g. "Help - The Beatles 2:19").
- play ( ): Opens a web browser and plays the song. The program will sleep for the duration of the song.

Create a `main` function (outside your Song class). We will use this function to test our class and will expand on it further in the next step.

You **must** test each method once you create it. You can write a helper functions (outside your class), to test it. Add the following code above your `main` function.

```
def testSongPrintInfo():
    s1 = Song("Help","https://www.youtube.com/watch?v=2Q_ZzBGPdqE", "02:19")
    s1.printInfo()
```

Then call the following code from inside your `main` function:

```
testSongPrintInfo()
```

Once your program works for `printInfo`, add the `play` method.

Add an additional test method entitled `testSongPlay()`, to test your `play` method.

PlaylistManager Class

Just having individual song objects is not all that helpful because we still need to collect them somehow and be able to play music. A `PlaylistManager` is a class that is responsible for keeping a list of the songs a user wants to store and play. Think of it like an iPod. You only need one of them, it initially comes out of the box with no songs stored in it. It has the ability to add a new song, play a song, shuffle songs, delete a song, list all the songs, etc. Our PlaylistManager will have only some of these abilities but you could easily keep working on this to make it have more and more of them.

Class **PlaylistManager**: Holds a list of Song objects in a playlist and provides methods for performing the operations on the playlist.

Attributes:

- song_list: list of `Song` objects

Contructor and Methods:

- constructor ( ): initializes an empty list.
- addSong (title, url, duration): creates a Song object and adds it to the playlist.
- printAll( ): prints all the Songs in the playlist each on a separate line preceded by a number. *Note: This should use the Song.print_info method.*
- playAll ( ): play all the Songs in the playlist in sequence

For example, the `print_all()` function may display:

1. Happy Holiday - Andy Williams 2:37
2. Help - The Beatles 2:19
3. Walkin Blues - Robert Johnson 3:02

Again, only write one method at a time and test it individually (add helper methods). It is more important to write and test printAll, before playAll because it will reveal basic bugs prior to trying to play songs.

To wrap up today's lab, finish the `main()` function below. It should create an instance of your `PlaylistManager`, allow the user to add songs until they say they're done, and then play the songs in the playlist. Use the methods you defined in your `PlaylistManager` and `Song` classes to do most of the work!

```
def main():
  # Instantiate PlaylistManager
  playlistMgr = PlaylistManager()
  keepGoing = ""
  while(keepGoing != "DONE"):
```

```
    # Ask for info about song
    title = input("Song title: ")
    url = input("YouTube video: ")
    duration = input("Video duration (MM:SS): ")

    ## TODO: Add the Song to the PlaylistManager

    keepGoing = input("Add another song? YES to continue, DONE to end: ")

  ## TODO: when done adding songs print all the songs
  ## TODO: after printing all songs, play them all.
if __name__ == "__main__":
  main()
```

Note: In this case, main does not have access to any of the song objects. This is an example of encapsulation.

Submit your answers to the Short Answer questions and your lab13p1.py, lab13p2.py, and PlayListManger codes on Canvas.