Patrick Moynihan
May 19, 2024
Course: Foundations of Programming: Python
Assignment: 06
GitHub link: https://github.com/Pyronious/IntroToProg-Python-Mod06

Exercise: Functions, Classes and Methods

Contents

Introduction 2
Data Preparation 2
  Constants 2
  Variables 3
Defining Classes 3
  FileProcessor 3
    read_data_from_file() 4
    write_data_to_file() 6
  IO 6
    output_menu() 7
    input_menu_choice() 7
    output_student_courses() 8
    input_student_data() 8
    output_error_messages() 9
Main logic 10
  Loading data from a JSON file 10
  Main Menu Loop 11
    Option 1: Register a student for a course 11
    Option 2: Show current data 11
    Option 3: Save the data to a file 12
    Option 4: Exit the program 12
  Summary 13

Introduction

The goal of this exercise is to learn about working with functions, classes, and methods. The functionality of our program is the same as last week. We're refactoring the code so that it is organized by separation of concerns. This will be accomplished through the implementation of classes. Our classes will represent features such as File Handling, Input/Output, and Error Messaging. Each class will contain methods which can be used to manipulate data.

Classes and methods allow us to keep re-usable code in a single easy-to-maintain location. Then whenever we need the functionality in our code, we can simply call the method.

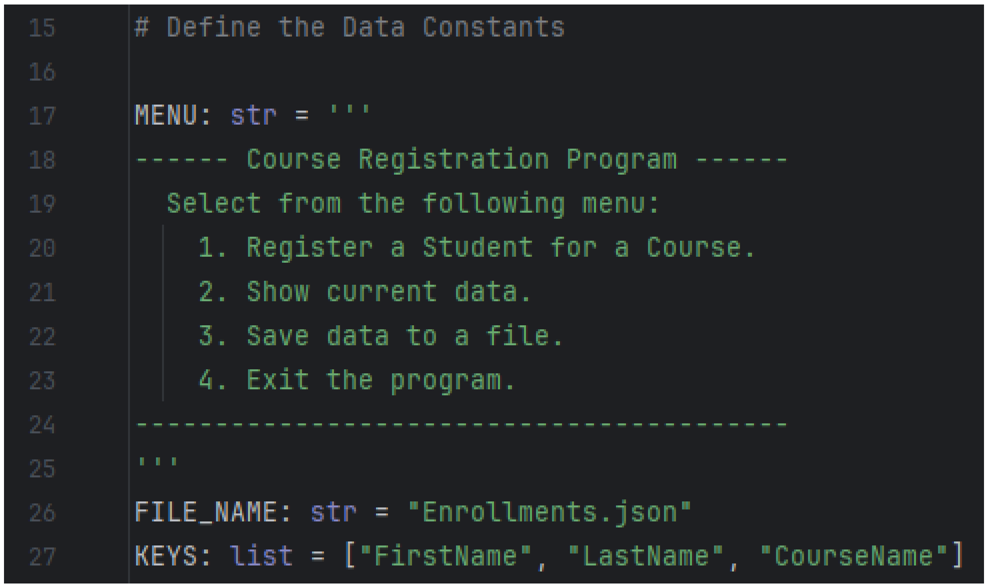
This is super convenient, especially if we want to change the way something works in the future. For example, if someday we want to have all our error messages sent as a push notification to our phone, we can simply update the Error Messaging method to use push notifications, and all error messages in our code will inherit that feature since they call our upgraded method.

Data Preparation

Our first step is to define the constants and variables we will be using throughout the script.

Constants

We start by declaring our constants. Nothing new this week.

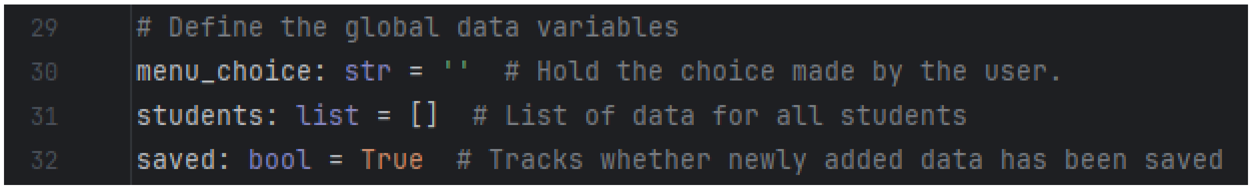


```python
# Define the Data Constants

MENU: str = '''
------ Course Registration Program ------
  Select from the following menu:
    1. Register a Student for a Course.
    2. Show current data.
    3. Save data to a file.
    4. Exit the program.
-----------------------------------------
'''
FILE_NAME: str = "Enrollments.json"
KEYS: list = ["FirstName", "LastName", "CourseName"]
```

Figure 1: Declaring constants.

Variables

We're using considerably fewer global variables this week. Just three:



```
# Define the global data variables
menu_choice: str = ''  # Hold the choice made by the user.
students: list = []  # List of data for all students
saved: bool = True  # Tracks whether newly added data has been saved
```

Figure 2: Declaring global variables.

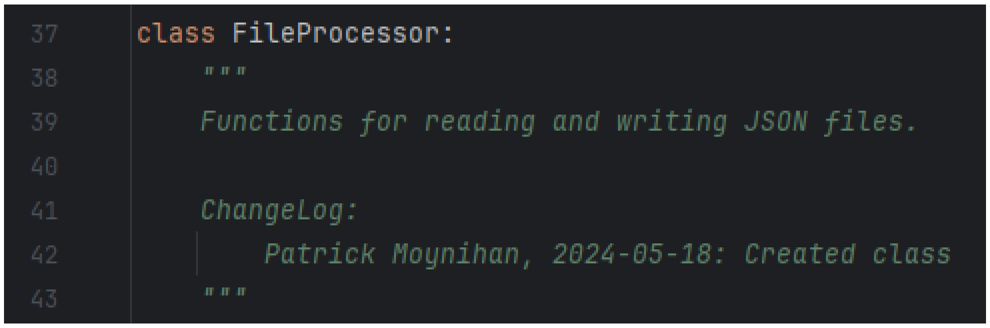
We've moved many of our other variables inside of the classes and methods we'll be using this week, as it is not necessary for those variables to have global scope.

Defining Classes

In their simplest form, classes are collections of functions which can be used to create reusable code. Classes can also be used to create objects, which can contain both data and methods for manipulating data. We won't be using classes as objects this week. Instead, we'll just be calling the functions inside of the classes. Since we're not using objects, we can only call the methods that are identified as static methods using the **@staticmethod** decorator. Static methods are available even without first instantiating an object from the class.

FileProcessor

We'll start by declaring our first class, **FileProcessor**.



```
class FileProcessor:
    """
    Functions for reading and writing JSON files.

    ChangeLog:
        Patrick Moynihan, 2024-05-18: Created class
    """
```

Figure 3: Declaring a class.

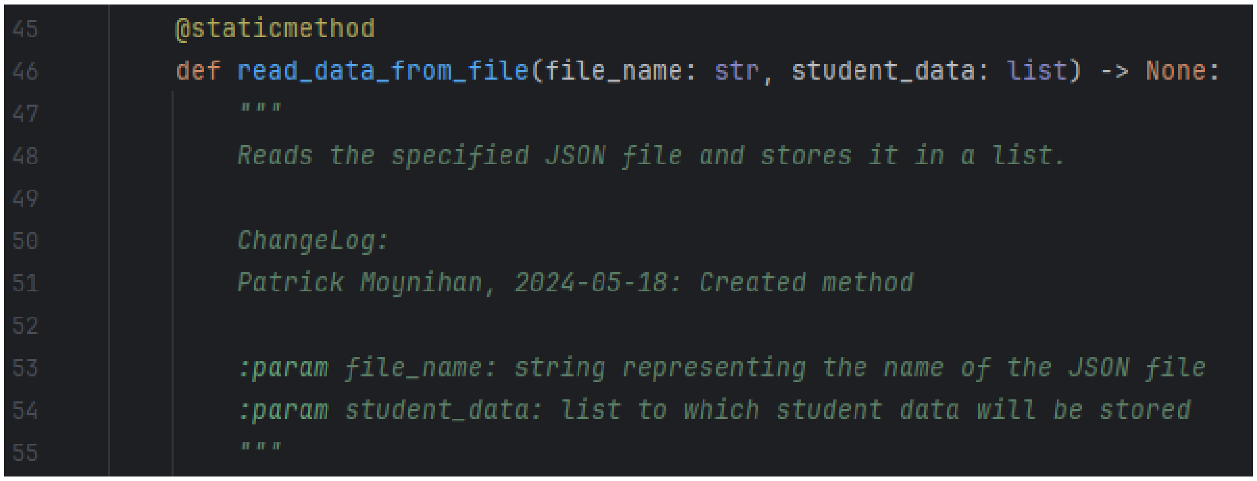
Our class definition is simple: we just give it a name and add a documentation string. The documentation string will allow PyCharm (or other IDEs) to offer pop-up help with information about the class when the user hovers their mouse over a call to this class from somewhere else in the code.

Inside of this class we'll declare 2 methods:

- read_data_from_file
- write_data_to_file

read_data_from_file()

Our first method inside the **FileProcessor** class is **read_data_from_file()**. The functionality of this method is the same as last week's assignment when we loaded the JSON data from a file on disk. We're going to move all that code inside this method, and then set up our method arguments and return values so that the code is more flexible and re-usable.



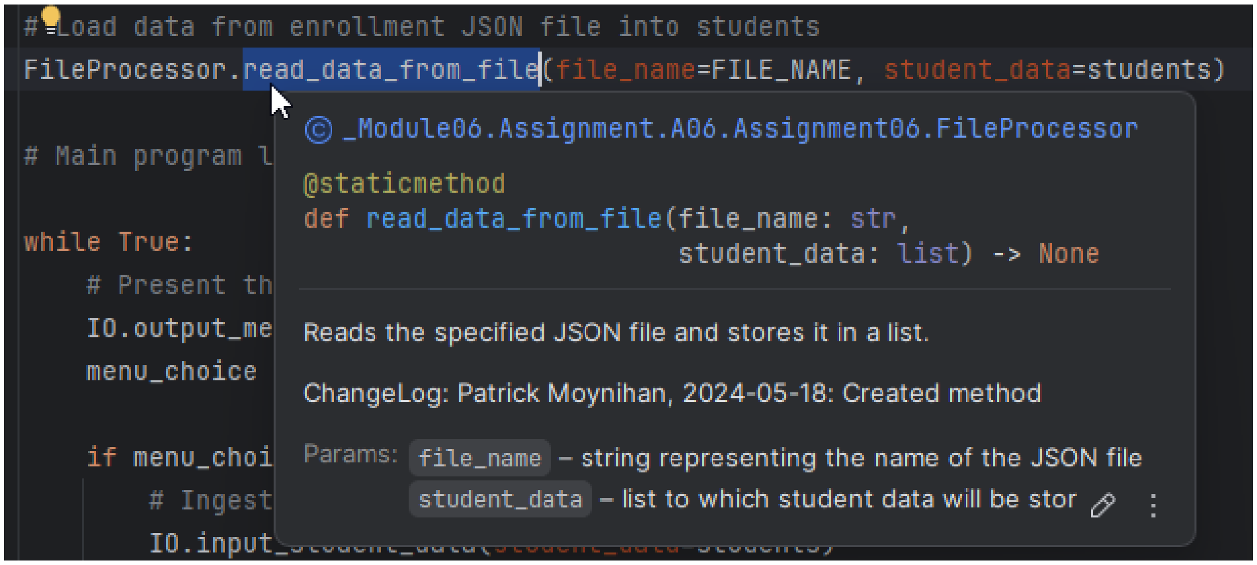
```
45      @staticmethod
46      def read_data_from_file(file_name: str, student_data: list) -> None:
47          """
48          Reads the specified JSON file and stores it in a list.
49
50          ChangeLog:
51          Patrick Moynihan, 2024-05-18: Created method
52
53          :param file_name: string representing the name of the JSON file
54          :param student_data: list to which student data will be stored
55          """
```

Figure 4: Declaring a method.

We start with the @staticmethod decorator on line 45, which allows us to access this method directly without first instantiating an object of type FileProcessor. I suspect we'll get into that kind of thing in future assignments.

Line 46 declares the method name, as well as the arguments it expects to receive. This method expects a string containing the file name we want to open (**file_name**), and a list to store the loaded data in (**student_data**). The **-> None:** at the end of the declaration indicates that this method does not provide a return value.

Lines 47-55 are our documentation string. On lines 53 and 54 we document what the parameters of the method are. This allows PyCharm (and other IDEs) to offer help text when working with the method.



```
#Load data from enrollment JSON file into students
FileProcessor.read_data_from_file(file_name=FILE_NAME, student_data=students)

# Main program l

while True:
    # Present th
    IO.output_me
    menu_choice

    if menu_choi
        # Ingest
        IO.input_
```

```
_Module06.Assignment.A06.Assignment06.FileProcessor
@staticmethod
def read_data_from_file(file_name: str,
                        student_data: list) -> None
```

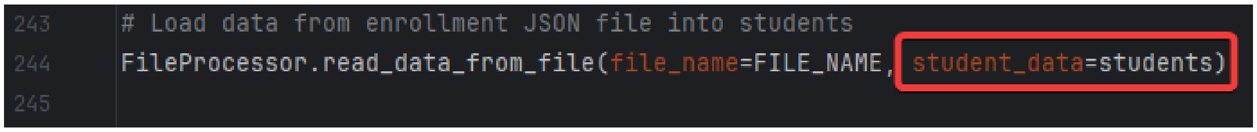
Reads the specified JSON file and stores it in a list.

ChangeLog: Patrick Moynihan, 2024-05-18: Created method

Params: file_name – string representing the name of the JSON file
student_data – list to which student data will be stor

Figure 5: How PyCharm uses the documentation string to provide contextual help.

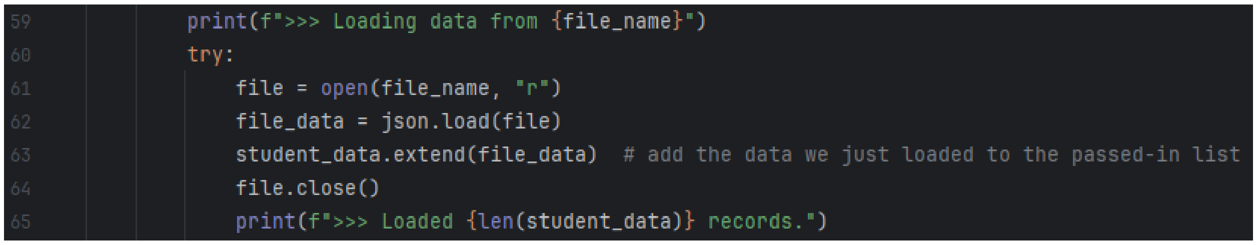
The only real change to the loading code this week is related to how we store the data in the global variable **students**. It's poor practice to directly assign values to global variables inside a method because it makes the method less portable/reusable in other projects. So instead, when we call the method later on we'll pass in the global variable **students** as an argument:



```
# Load data from enrollment JSON file into students
FileProcessor.read_data_from_file(file_name=FILE_NAME, student_data=students)
```

Inside our method, we can use the local variable **student_data**, which contains a reference to the global variable **students**. As a result, when we make changes to values inside **student_data**, the values inside **students** will also change.

We can see that happening inside the code that loads the file:



```
59          print(f">>> Loading data from {file_name}")
60          try:
61              file = open(file_name, "r")
62              file_data = json.load(file)
63              student_data.extend(file_data)  # add the data we just loaded to the passed-in list
64              file.close()
65              print(f">>> Loaded {len(student_data)} records.")
```

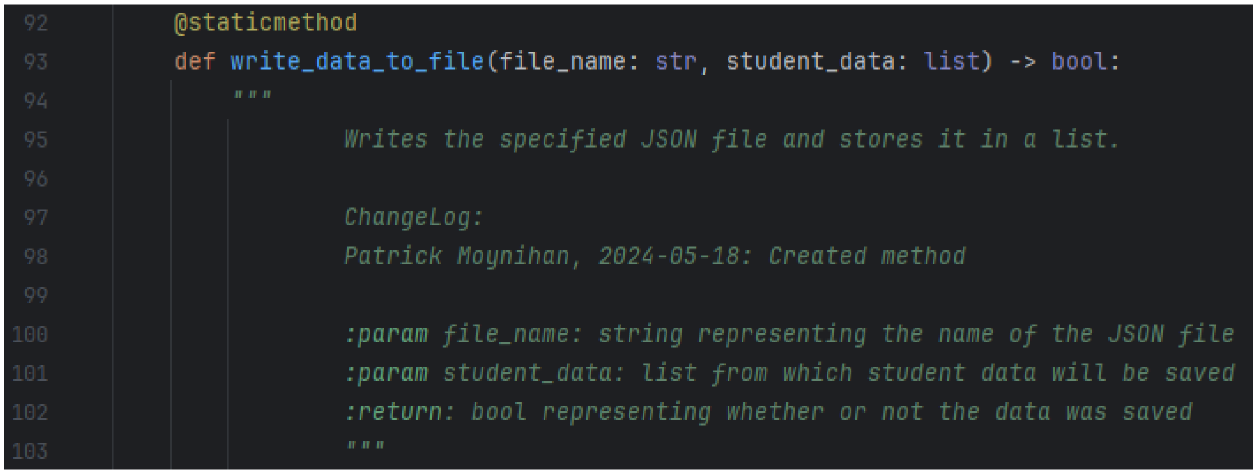
Figure 6: File loading code inside the load_data_from_file method.

Line 62 loads the JSON data from a file into a temporary variable **file_data**. Then Line 63 calls the **.extend()** method on **student_data** to append the **file_data** to **student_data**.

It's important to note that because **student_data** contains a reference to **students**, what actually ends up happening here is that the **file_data** is appended to **students**.

write_data_to_file()

This method contains our file writing code from last week. We'll declare it like so:



```
    @staticmethod
    def write_data_to_file(file_name: str, student_data: list) -> bool:
        """
            Writes the specified JSON file and stores it in a list.

            ChangeLog:
            Patrick Moynihan, 2024-05-18: Created method

            :param file_name: string representing the name of the JSON file
            :param student_data: list from which student data will be saved
            :return: bool representing whether or not the data was saved
            """
```

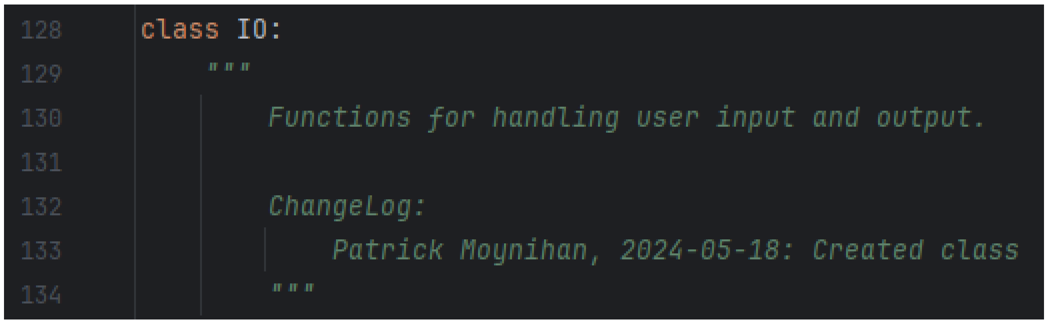
Figure 7: Declaring the FileProcessor.write_data_to_file method.

The method expects two parameters. **file_name** is used to tell the method the name of the file we want to write to. **student_data** is used to provide the list of data we want to write.

We follow that with our standard documentation string.

IO

Our **IO** class will handle retrieving user input and presenting output to the user. We declare the **IO** class with a documentation string as we did with the **FileProcessor** class.



```
class IO:
    """
        Functions for handling user input and output.

        ChangeLog:
            Patrick Moynihan, 2024-05-18: Created class
        """
```

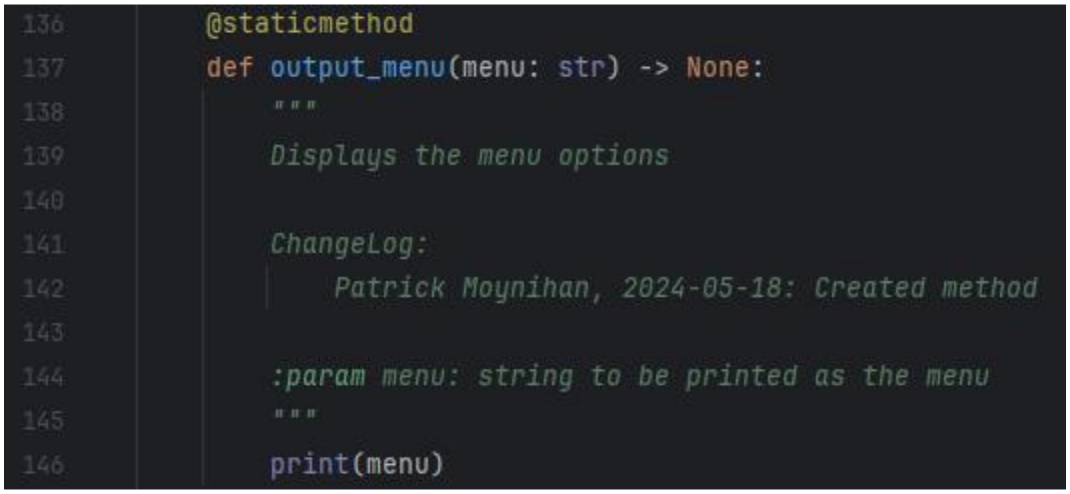
Figure 8: Declaring the IO class.

Inside of this class we'll implement five methods:

- output_menu
- input_menu_choice
- output_student_courses
- input_student_data
- output_error_messages

output_menu()

This method prints the menu string to the terminal.



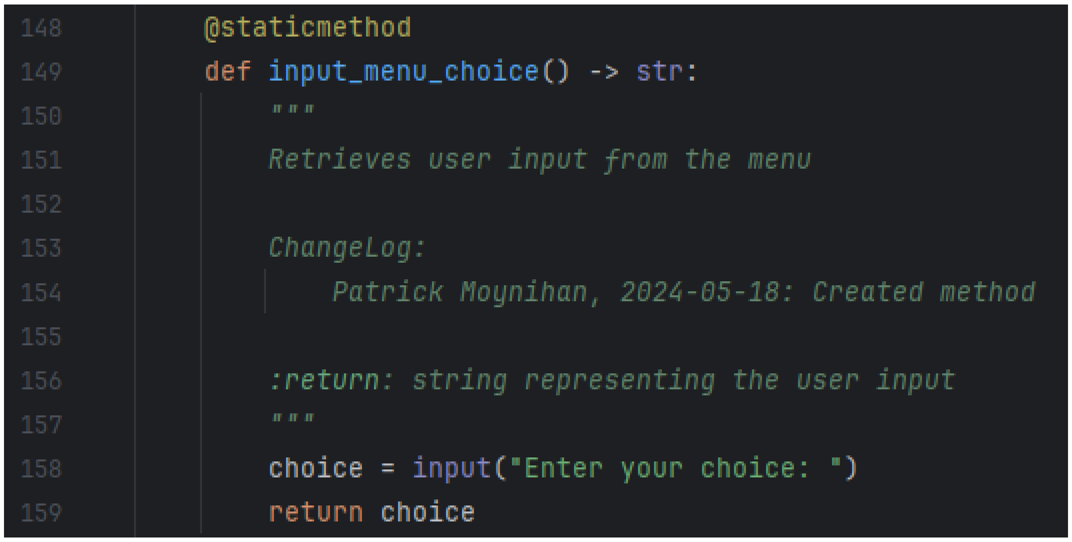
```
136     @staticmethod
137     def output_menu(menu: str) -> None:
138         """
139         Displays the menu options
140
141         ChangeLog:
142             Patrick Moynihan, 2024-05-18: Created method
143
144         :param menu: string to be printed as the menu
145         """
146         print(menu)
```

Figure 9: IO.output_menu method.

The string that is printed on line 146 is passed in via the **menu** argument.

input_menu_choice()

This method gets user input and returns it as a string.

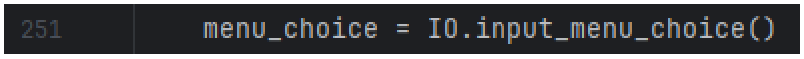


```
148     @staticmethod
149     def input_menu_choice() -> str:
150         """
151         Retrieves user input from the menu
152
153         ChangeLog:
154             Patrick Moynihan, 2024-05-18: Created method
155
156         :return: string representing the user input
157         """
158         choice = input("Enter your choice: ")
159         return choice
```

Figure 10: IO.input_menu_choice_method.

This method uses a return value. Line 159 returns the value of **choice**.

We'll use the return value later on in our code to retrieve the user input:

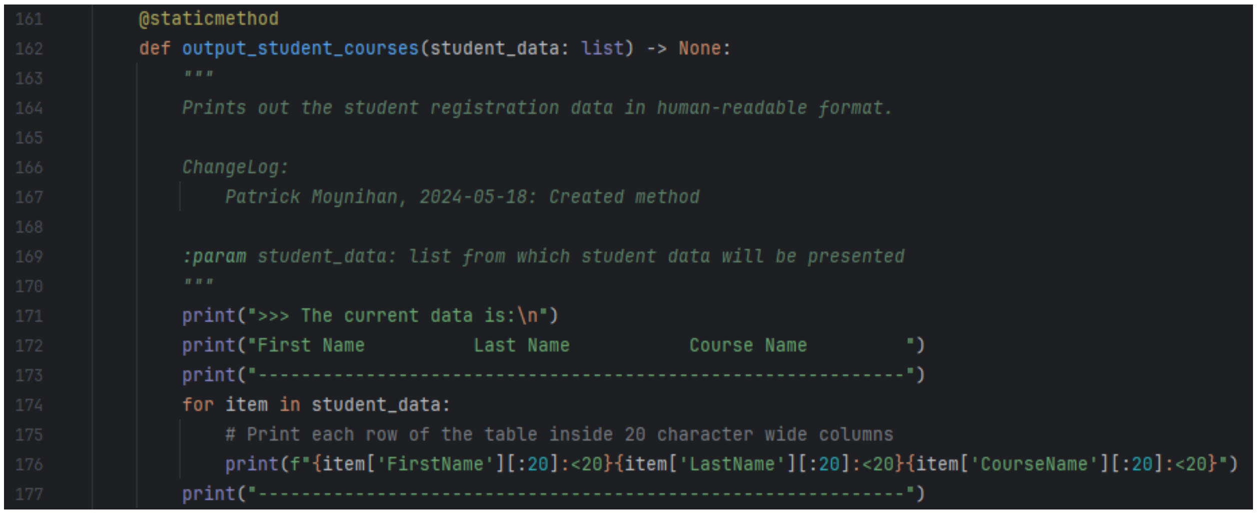


```
        menu_choice = IO.input_menu_choice()
```

By using the assignment operator **=**, we are setting the value of **menu_choice** to whatever gets returned by the method **IO.input_menu_choice**.

output_student_courses()

This method prints the list of registered users to the screen:



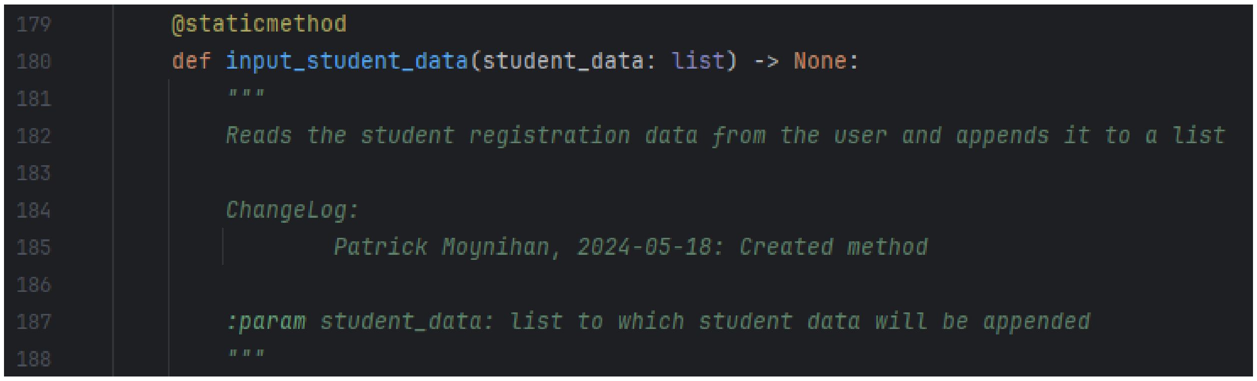
```
    @staticmethod
    def output_student_courses(student_data: list) -> None:
        """
        Prints out the student registration data in human-readable format.

        ChangeLog:
            Patrick Moynihan, 2024-05-18: Created method

        :param student_data: list from which student data will be presented
        """
        print(">>> The current data is:\n")
        print("First Name          Last Name           Course Name         ")
        print("------------------------------------------------------------")
        for item in student_data:
            # Print each row of the table inside 20 character wide columns
            print(f"{item['FirstName'][:20]:<20}{item['LastName'][:20]:<20}{item['CourseName'][:20]:<20}")
        print("------------------------------------------------------------")
```

Figure 11: IO.output_student_courses method.

The list that is to be printed is passed in via the argument **student_data**. The rest of the code is the same as last week.

input_student_data()

This method gets a new student registration entry from the user and appends it to the passed-in list.

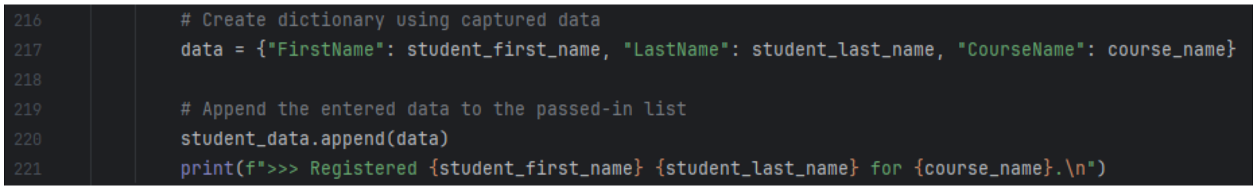


```
    @staticmethod
    def input_student_data(student_data: list) -> None:
        """
        Reads the student registration data from the user and appends it to a list

        ChangeLog:
                Patrick Moynihan, 2024-05-18: Created method

        :param student_data: list to which student data will be appended
        """
```

Figure 12: Declaring IO.input_student_data.

We're again passing in a reference to a list (**student_data**) as an argument to the method. This will allow the method to perform operations on the list that we pass in.

The rest of the code in this method is largely the same as last week, with one notable exception:



```
216         # Create dictionary using captured data
217         data = {"FirstName": student_first_name, "LastName": student_last_name, "CourseName": course_name}
218 
219         # Append the entered data to the passed-in list
220         student_data.append(data)
221         print(f">>> Registered {student_first_name} {student_last_name} for {course_name}.\n")
```

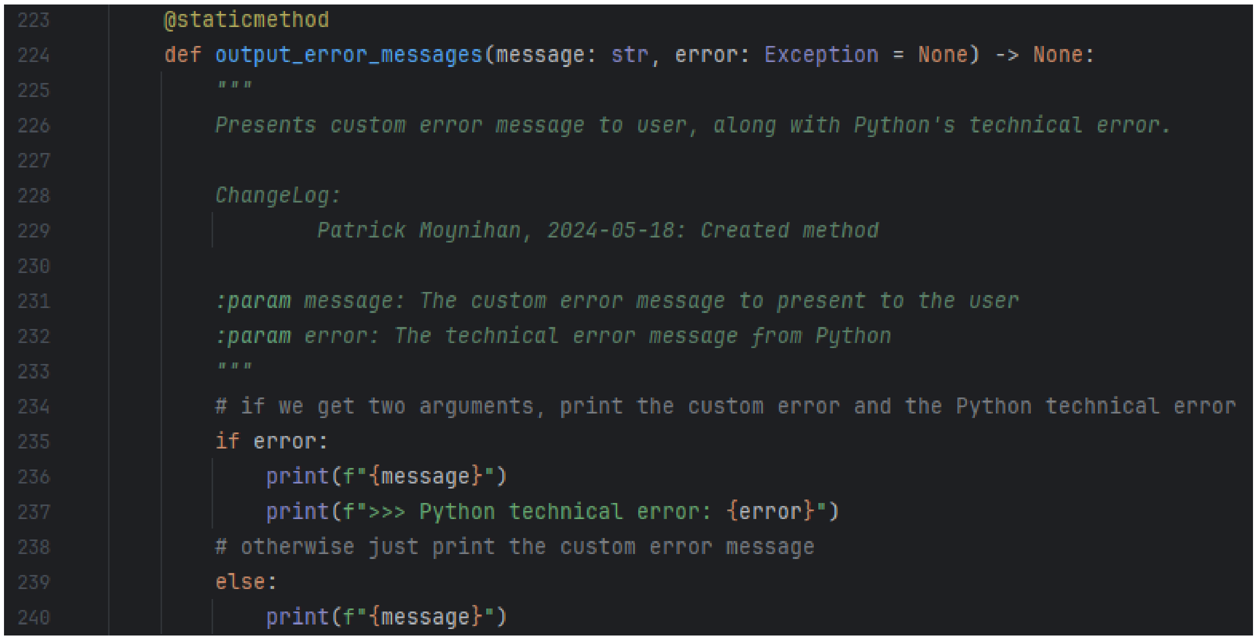
Figure 13: Assigning the registration data to the passed-in list.

We first collect the user data into a local variable **data** on line 217. Then we use the **.append()** method on **student_data** to append the data to the list that was passed in.

Again, we're doing it this way so that our method isn't hard-coded to the global variable **students**. Instead, we can pass **students** (or any other list we like) as an argument to the method. This makes the method more general-purpose and re-usable.

output_error_messages()

This method will be used to present error messages to the user.



```
223     @staticmethod
224     def output_error_messages(message: str, error: Exception = None) -> None:
225         """
226         Presents custom error message to user, along with Python's technical error.
227 
228         ChangeLog:
229                 Patrick Moynihan, 2024-05-18: Created method
230 
231         :param message: The custom error message to present to the user
232         :param error: The technical error message from Python
233         """
234         # if we get two arguments, print the custom error and the Python technical error
235         if error:
236             print(f"{message}")
237             print(f">>> Python technical error: {error}")
238         # otherwise just print the custom error message
239         else:
240             print(f"{message}")
```

Figure 14: IO.output_error_messages method.

This method takes two arguments, but one of them is optional. The string argument **message** is required because it does not have a default value. However the **error** argument has a default value of **None**. That means we can call this method with either one or two arguments.

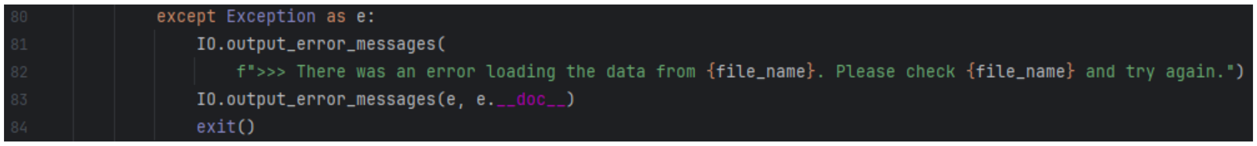
Line 235 checks to see if the method was called with a 2nd argument. If a 2nd argument was not passed, **error** will contain the default value of **None** and therefore **if error:** evaluates to False. But if a 2nd argument was passed in, **if error:** evaluates to true.

The reason for doing this is that sometimes we just want to present the user with a simple custom error message (passed in via **message**). But other times we want to present the user with both a custom error message and a Python-generated exception message (the "Technical Error", passed in as **error**).

This is Python's way of emulating an overloaded method, as Python does not support overloaded functions/methods directly.

All of the error handling logic in our methods has been updated to use this new method for displaying errors.

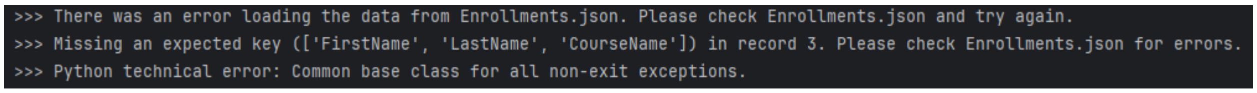
Here's an example that shows the difference between passing a single argument and both arguments:



```
80          except Exception as e:
81              IO.output_error_messages(
82                  f">>> There was an error loading the data from {file_name}. Please check {file_name} and try again.")
83              IO.output_error_messages(e, e.__doc__)
84              exit()
```

Figure 15: Passing arguments to the method.

Line 81 passes a single argument with a custom error message. Line 83 passes dual arguments with the **e** being the error message from Python and **e.__doc__** being a more detailed error from Python.



```
>>> There was an error loading the data from Enrollments.json. Please check Enrollments.json and try again.
>>> Missing an expected key (['FirstName', 'LastName', 'CourseName']) in record 3. Please check Enrollments.json for errors.
>>> Python technical error: Common base class for all non-exit exceptions.
```

Figure 16: Error messages from single and dual argument calls to the method.

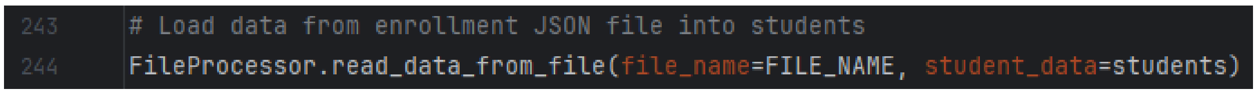
The first line is the output from our single-argument call to **IO.output_error_message**.

The second and third lines are the output from our double-argument call to **IO.output_error_message**.

Main logic

Loading data from a JSON file

We start by loading the data from a JSON file.



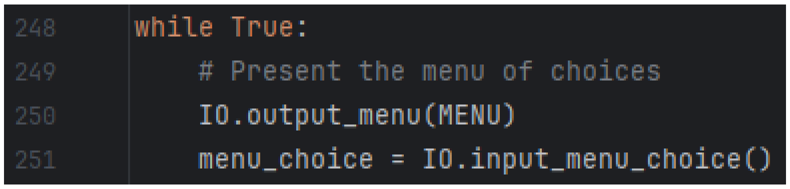
```
# Load data from enrollment JSON file into students
FileProcessor.read_data_from_file(file_name=FILE_NAME, student_data=students)
```

We did all the work for this already in our FileProcessor.read_data_from_file method, so all we have to do is call it and pass in the arguments **file_name** (the name of the file we want to read) and

student_data (the list we want to write the data to). In this case we pass in **FILE_NAME** and **students** as our arguments. The method will load the file and put the results in **students**.

Main Menu Loop

Now that the data is loaded, we enter the main menu loop. This week we're using our new methods to present the menu and get the user's selection:

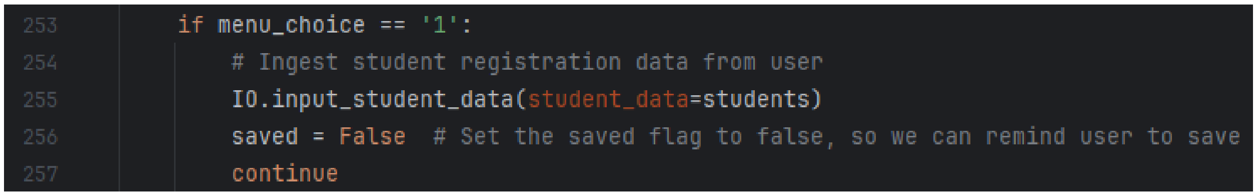


```python
while True:
    # Present the menu of choices
    IO.output_menu(MENU)
    menu_choice = IO.input_menu_choice()
```

Figure 17: Setting up the main menu loop.

Option 1: Register a student for a course

We'll just call our method **IO.input_student_data** here to retrieve the user data. We pass in **students** as the argument so that the method can update that list with the new data for us.



```python
253     if menu_choice == '1':
254         # Ingest student registration data from user
255         IO.input_student_data(student_data=students)
256         saved = False  # Set the saved flag to false, so we can remind user to save
257         continue
```

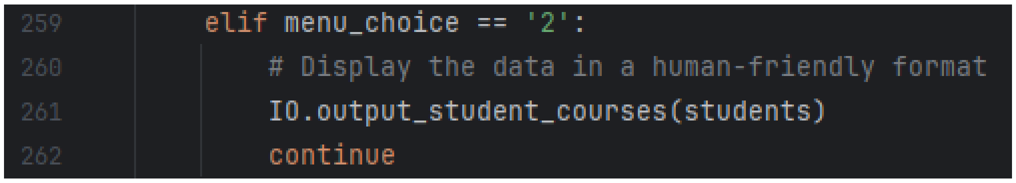
Figure 18: Handling menu option 1.

Line 256 sets our **saved** flag to **False** so that later on we can remind the user to save before exiting the program.

Line 257 instructs the while loop block we're in (the main menu loop) to continue from the top.

Option 2: Show current data

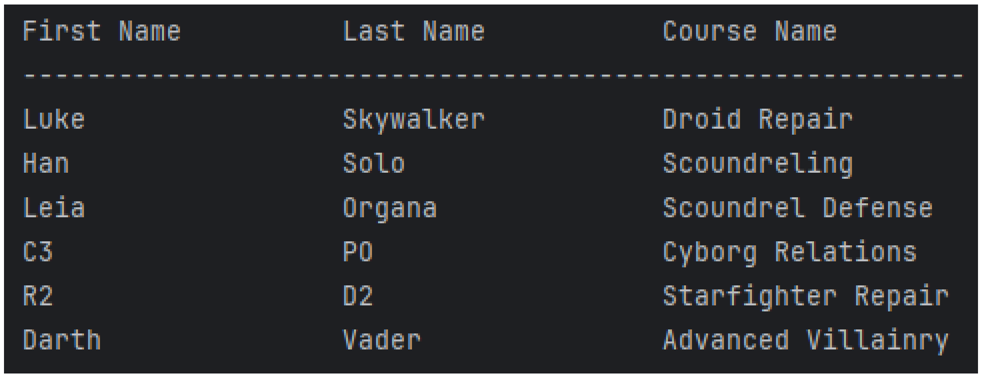
We've moved all the code for this into the method **IO.output_student_courses** so we can just call that here.



```python
    elif menu_choice == '2':
        # Display the data in a human-friendly format
        IO.output_student_courses(students)
        continue
```

Figure 19: Handling menu option 2.

We pass **students** as the argument, which results in this output:

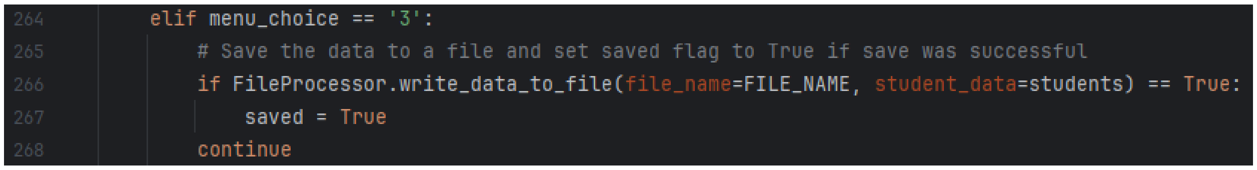


```
First Name          Last Name           Course Name
------------------------------------------------------------
Luke                Skywalker           Droid Repair
Han                 Solo                Scoundreling
Leia                Organa              Scoundrel Defense
C3                  PO                  Cyborg Relations
R2                  D2                  Starfighter Repair
Darth               Vader               Advanced Villainry
```

Figure 20: Example human readable output for option 2.

Option 3: Save the data to a file

We've moved all the save code into the method **IO.write_data_to_file**, so we can just call that here:



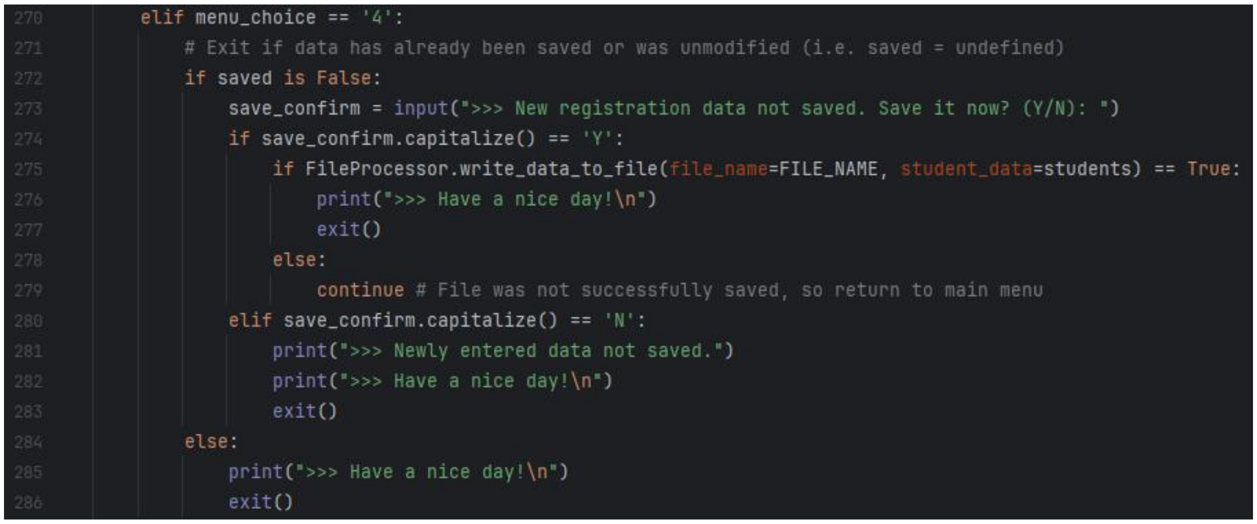
```python
    elif menu_choice == '3':
        # Save the data to a file and set saved flag to True if save was successful
        if FileProcessor.write_data_to_file(file_name=FILE_NAME, student_data=students) == True:
            saved = True
        continue
```

Figure 21: Handling menu option 3.

We're doing something a bit different here. Our method **IO.write_data_to_file** returns a value of **True** if it was able to successfully save the file. Otherwise it returns a value of **False**. We use that information to set the correct value of the global variable **saved**, which tracks whether the user's data has been saved. Basically, if the method returns **True**, we know that it successfully saved the file and we can set **saved** to **True**.

Option 4: Exit the program

When the user opts to exit the program, we first check to see if they have any unsaved registration data. If so, we ask them if they'd like to save it before exiting.

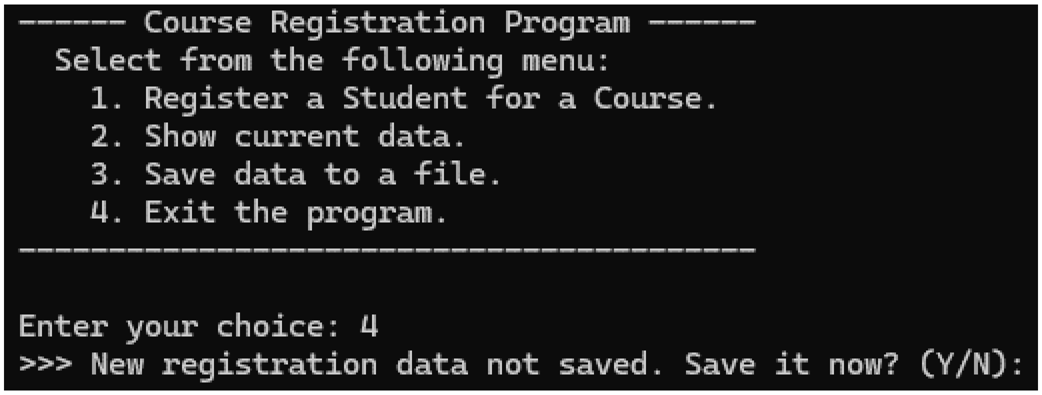


```python
        elif menu_choice == '4':
            # Exit if data has already been saved or was unmodified (i.e. saved = undefined)
            if saved is False:
                save_confirm = input(">>> New registration data not saved. Save it now? (Y/N): ")
                if save_confirm.capitalize() == 'Y':
                    if FileProcessor.write_data_to_file(file_name=FILE_NAME, student_data=students) == True:
                        print(">>> Have a nice day!\n")
                        exit()
                    else:
                        continue # File was not successfully saved, so return to main menu
                elif save_confirm.capitalize() == 'N':
                    print(">>> Newly entered data not saved.")
                    print(">>> Have a nice day!\n")
                    exit()
            else:
                print(">>> Have a nice day!\n")
                exit()
```

Figure 22: Warning the user that their data hasn't been saved.

Last week we handled this a little bit differently because it didn't make sense to write the file saving code twice. If they wanted to save, they had to return to the main menu to save their work using option 3. It was fine, but we can make it more convenient.

This week, we can leverage the **IO.write_data_to_file** method to save their work from right here inside the option 4 handler, which is much more convenient.



```
------ Course Registration Program ------
  Select from the following menu:
    1. Register a Student for a Course.
    2. Show current data.
    3. Save data to a file.
    4. Exit the program.
-----------------------------------------

Enter your choice: 4
>>> New registration data not saved. Save it now? (Y/N):
```

Figure 23: Example output of exit confirmation

Summary

In this assignment we learned:

1. How to declare classes
2. How to declare functions and methods
3. How to organize code by the separation of concerns pattern
4. How to pass arguments to functions & methods
5. How to manipulate data in global variables from functions and methods via reference
6. How to emulate the behavior of overloaded functions & methods in Python