Stefanie Taushanoff
May 28, 2025
IT FDN 100 A (Spring 25)
Assignment 06

Functions & Classes

GitHub repository link: https://github.com/ST-04261/IntroToProg-Python-Mode06

Introduction

Assignment 06 begins to look like real code by introducing functions and classes. Functions allow us to compartmentalize our program into small, reusable modules. Thinking modularly forces you to really break down your code and make it organized and sensible, then provides you with debugged modules that can be reused elsewhere.

Classes are an abstraction that includes functions, variables and constants. This keeps closely related functions and variables together. Multiple classes can be used in a single program, and can interact with each other through proper use of parameters and calls.

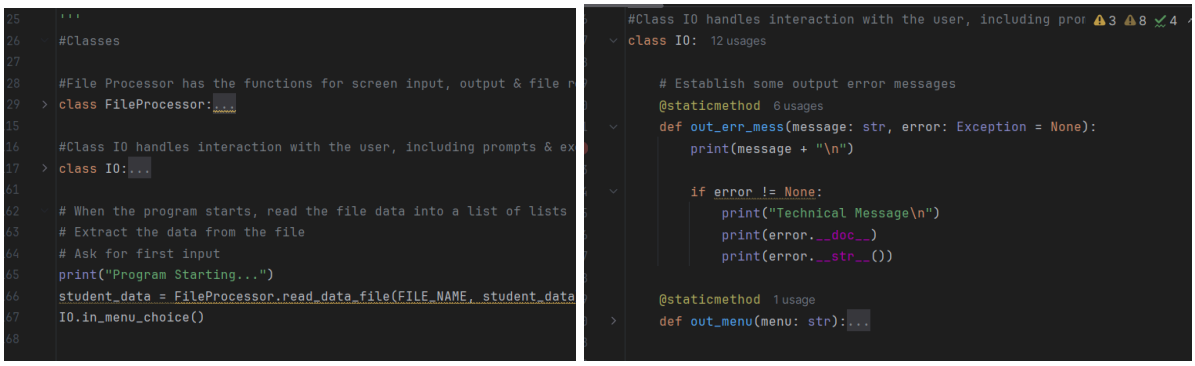

Figure 1: Using classes and functions, our nearly 200-line program looks a lot cleaner in PyCharm (left) allowing us to focus on the code we're working on (right)

To use functions and classes properly we covered parameters, arguments, calls, and global v local variables.

Functions

A function is a modular unit of code that performs a defined task. Until this assignment, our program was one big unoptimized function. It had a lot of repetition, particularly in the exception handling where each menu option listed out its own unique exceptions. It also had many variables, lists, and dictionaries to keep track of new data as it came in.

In this code, we compartmentalize the tasks into input, output, read and write, plus some exception handling. The code outside of these functions (minus some variable declarations at the top) is reduced

to three lines, with only two of them being strictly necessary (Fig 2):

```
# When the program starts, read the file data into a list of lists (table)
# Extract the data from the file
# Ask for first input
print("Program Starting...")
student_data = FileProcessor.read_data_file(FILE_NAME, student_data)
IO.in_menu_choice()
```

Figure 2: The program only needs two steps to get started.

We print Program Starting as a courtesy to the user. The program then reads a JSON file (something it only needs to do once per program execution). Then we call in_menu_choice to let the user get started by making their initial choice. By taking the four menu options and modularizing them, we'll also get rid of the while loop that, to this point, has been key to the program's functioning.

Note that in_menu_choice takes no arguments and returns nothing. Its purpose is to ask us a question and use our response to call upon the next desired function. It also contains a check to make sure your selection is valid, so functions called in the future can be assured they're getting good information (Fig 3).

```
@staticmethod  5 usages
def in_menu_choice():
    print(MENU)
    menu_choice: str = input("What would you like to do: ")

    if menu_choice in ['1', '2', '3', '4']:
        IO.out_menu(menu_choice)
        pass
    else:
        IO.out_err_mess("Invalid choice. Please choose from the menu.")
        IO.in_menu_choice()
```

Figure 3: in_menu_choice displays the menu and asks for user input. Should the user select an invalid choice, they get lighty chastised and we run the function again.

Now that we have the user input, we need to communicate that to other functions. Out_menu requires an argument (menu_choice) to determine its actions. Out_menu has defined parameters (it needs an input) which is why it needs to be passed this argument (FIg 4).

Like in_menu_choice, out_menu calls another function and passes a single argument, student_data. Student_data is a global variable, a list of dictionaries. We are reduced to three global variables, that is

three variables declared at the top of the program that all functions and classes can see and edit. Everything else is nested in the functions, and are therefore more ephemeral. When the function is not running, their local variables don't stick around. By abstracting things this way, we can get rid of globally defined temp variables instead only using them when we need them (Fg 5).

Figure 4: The parameters are in the parenthesis. When you call this function, it expects to see a valid string. If we didn't check the validity of the input in in_menu(), out_menu() could fail to run properly.

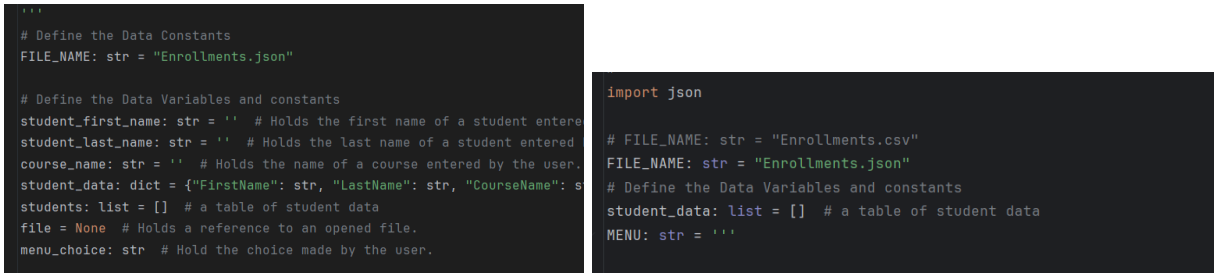

Figure 5: Eight variables initialized first thing in assignment 5 (left), vs three in assignment 6 (right).

The data processing functions work mostly the same way, taking arguments and performing tasks. One function, read_data_file is special, it needs to provide its data to the rest of the program. The information from the file is saved in the variable student_data and to pass that from the function to the variable, we use the return command (Fig 6):

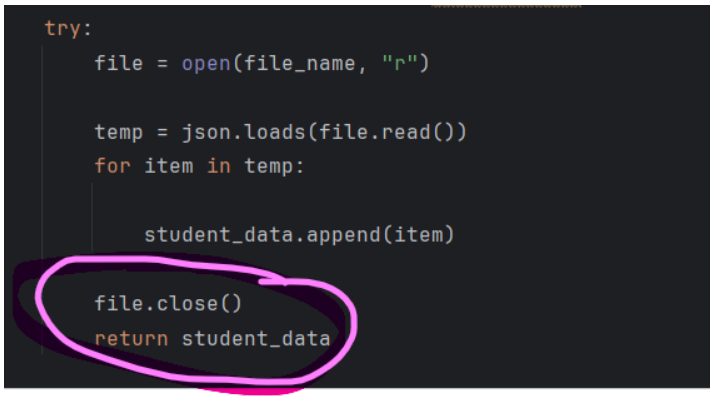

Figure 6: Rretun tells the function to pass its results to a specific place when you're not calling another function.

Classes

All of these functions are divided into two classes: IO and FIleProcessing. The IO class contains functions that either request input or display output to the reader. The FileProcessing class is responsible for executing the tasks listed in the Menu (Fig 7).

```
---- 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.
----------------------------------------
'''
#Classes

#File Processor has the functions for screen input, output & file reading & writing
class FileProcessor:...

#Class IO handles interaction with the user, including prompts & exceptions
class IO:...
```

Figure 7: The code is as detailed as ever, but by sorting into Classes it is prevented from becoming unreadable spaghetti.

When you put a function inside a class, it is no longer directly accessible outside of that class. When calling those functions, be sure to tell your program where to find them using the format of ClassName.FunctionName(args) (Fig 8):

```
@staticmethod  1 usage
def out_menu(menu: str):

    if menu == "1":
        print("Selection was " + menu)
        FileProcessor.in_studata(student_data)

    elif menu == "2":
        print("Selection was " + menu)
        FileProcessor.out_stucourses(student_data)
```

Fig 8: Functions in Class IO call upon functions in Class FileProcessor

Summary

Functions and classes allow us to write larger, more sophisticated programs using the concepts of modularity and repeatability. With a collection of functions or classes, code can be easily contained, debugged and reused. It's the difference between assembling a puzzle vs making an oil painting from scratch each time. Assignment 06 was definitely easier to debug than Assignment 05, and following the trail of arguments helps you understand exactly what the code is doing.

It is helpful to divide any program into blocks like Presentation, Logic or Data Storage. In our example, since we're only storing a little input, we have two concerns: Presentation and Processing. These formed our classes. Nearly all of the student registration program was able to be divided into components that fit within those concerns.

The added danger comes in defining the variables. Global variables should not share names with local variables. There is risk of passing the wrong arguments, or invalid arguments, to functions. It is good practice to minimize the modification of variables within the function, rather use the return function.