

Sophia Brooks

05/27/2025

IT FDN 110 A

Assignment #6

<https://github.com/SEBrooks25/IntroToProg-Python-Mod06>

# Functions

## Introduction

Understanding how to structure and organize code is a critical step in developing clear, reliable, and professional Python programs. As new tools and concepts are introduced, such as global and local variables, parameters, return values, and classes, developers gain more control over how data is managed and how different parts of a program interact. Each concept plays a key role in improving encapsulation and promoting modularity, both of which help keep code readable and adaptable. Parameters allow functions to be flexible and reusable, while return values help transfer results cleanly between parts of a program without relying on global data. Classes group related behavior and data into organized units, making it easier to manage complexity as a program grows. The use of local variables encourages tighter control over data scope, reducing the chance of unintended interactions between parts of the program. The Separation of Concerns design pattern takes this further by dividing a program into distinct layers for logic, data, and presentation. This makes it easier to update or troubleshoot a specific area without affecting the rest of the program. As these tools are combined, code naturally becomes more modular, scalable, and easier to collaborate on in group settings. Building with structure in mind from the start leads to better practices and more maintainable programs over time.

## Organizing Code

As new concepts and tools are introduced, scripts naturally become longer and more complex, making it essential to follow well-structured and logical methods for organizing code. Organizing code into functions is one of the most important techniques for improving clarity, reusability, and maintainability. A function, a named block of code that performs a specific task, can be reused throughout a program without repeating the same lines of code. By grouping related statements into functions, the program becomes easier to read and debug, especially as it grows in size. This modular approach also allows each function to focus on a single responsibility. Placing all function definitions at the beginning of the script, after constants and variables, ensures that the functions are recognized before being called later in the main body of the code. When building a program incrementally, it's helpful to outline the structure of future functions using the `pass` keyword as a placeholder (Figure 1). This allows the script to run without errors even though the function's logic hasn't been written yet (Figure 2). The `pass` keyword signals that the function exists but will be completed later, which is useful for planning and testing code structure in early stages. As the program evolves, each `pass` can be replaced with the actual processing logic needed for that part of the script. Using functions in this way

keeps the code clean, organized, and easier to manage, especially when changes need to be made later.

```
1  def load_data(): 1 usage
2      pass # This function will later load data from a file
3
4  def process_data(): 1 usage
5      pass # This function will later process the data
6
7  def display_results(): 1 usage
8      pass # This function will later display the processed results
9
10 # Main body of the script
11 print("Program started")
12 load_data()
13 process_data()
14 display_results()
15 print("Program ended")
```

Figure 1: Shows example usage of the pass keyword in code.

```
C:\Users\Sophia\Documents\Python\PythonCourse\
Program started
Program ended

Process finished with exit code 0
```

Figure 2: Shows example usage of the pass keyword results.

## Global vs Local Variables

Variables can be classified as either global or local depending on where they are defined in a script. Global variables are declared outside of functions and can be accessed from anywhere in the program, making them useful for constants or shared settings like file names. Local variables, on the other hand, are defined inside a function and only exist during that function's execution. Because of their limited scope, local variables are ideal for temporary values that don't need to be reused elsewhere. When a variable is assigned inside a function, Python treats it as local unless the global keyword is used (Figure 3). If a local variable has the same name as a global one, it will shadow the global variable within that function, leaving the original global variable unchanged (Figure 4). This behavior can be a source of confusion if developers aren't careful about naming and scope. It's generally better to use local variables and pass data in and out of functions using parameters and return values. This makes code more modular, predictable, and easier to maintain. By keeping global variables to a minimum and using local ones wherever possible, programs become more organized and less prone to bugs.

```

1  # Global variable
2  message = "Hello from the global scope!"
3
4  def greet(): 1 usage
5      # Local variable with the same name as the global variable
6      message = "Hello from the local scope!"
7      print(message) # This prints the local variable
8
9  greet()
10 print(message) # This prints the global variable
11

```

Figure 3: Shows an example implementation of global vs local variables.

```

C:\Users\Sophia\Documents\Python\PythonCourse
Hello from the local scope!
Hello from the global scope!

Process finished with exit code 0

```

Figure 4: Shows the results of the global vs local variable script.

## Parameters

Parameters are variables defined in a function that allow it to receive input data when called. They act as placeholders for values the function needs to work with, making the function reusable with different inputs (Figure 5). This supports encapsulation by hiding the internal details while clearly specifying what information the function requires. Using parameters keeps code modular and easier to maintain, as functions don't rely on fixed data or global variables. They also improve flexibility by letting the same function handle many tasks with different arguments. The implementation of parameters help to organize data flow and keep programs clean and adaptable.

```

1  def greet(name): 2 usages
2      print(f"Hello, {name}!")
3
4  greet("Alice")
5  greet("Bob")

```

Figure 5: Shows example usage of parameter (name) in the *greet* function.

Default parameters and overloaded functions add more flexibility in how functions handle input. Default parameters give a preset value when no argument is provided, allowing simpler calls and reducing repetitive code (Figure 6). Overloaded functions let multiple versions of a

function share the same name but differ in their parameter types or numbers (Figure 7). Both features improve encapsulation by hiding complexity and letting related operations share a name. They enhance code organization and clarity by reducing the number of different function names. Together, these tools help write adaptable and maintainable code with flexible interfaces.

```
1 # Default parameter example
2 def describe_pet(pet_name, pet_type="dog"): 2 usages
3     print(f"I have a {pet_type} named {pet_name}.")
4
5 describe_pet("Buddy") # Uses default pet_type "dog"
6 describe_pet(pet_name="Whiskers", pet_type="cat") # Uses provided pet_type "cat"
```

Figure 6: Shows implementation of a default parameter for *pet\_type*, making it optional.

```
1 # Simulating overloaded functions with variable arguments
2 def multiply(a, b=None): 2 usages
3     if b is None:
4         return a * a # If one argument, square it
5     else:
6         return a * b # If two arguments, multiply them
7
8 print(multiply(4)) # Output: 16 (4 squared)
9 print(multiply(a=4, b=5)) # Output: 20 (4 times 5)
```

Figure 7: Shows how one function name can do different things depending on number of inputs.

Returning data by reference means a function returns a direct link to existing data instead of copying it. This improves performance, especially for large data, by avoiding costly copying operations. It also allows callers to modify the original data through the returned reference, adding flexibility. Returning by reference supports encapsulation by controlling data access but requires clear design to avoid unintended side effects. It helps organize code by reducing redundant data handling and simplifying data sharing between functions. When used carefully, this technique leads to faster, clearer, and more maintainable programs.

```
1 def get_list(): 1 usage
2     my_list = [1, 2, 3]
3     return my_list # Returns a reference to the list
4
5 numbers = get_list()
6 numbers.append(4)
7 print(numbers) # Output: [1, 2, 3, 4]
```

Figure 8: Shows the function as it returns a reference to *my\_list* and how modification outside the function changes the original data.

## Return Values

Return values are the results that functions send back after completing their work. When a function returns a value, it can be stored in a variable for use later (Figure 9). This makes it easy to pass information between different parts of the code without relying on global variables. Returning values supports encapsulation by keeping the function's internal details hidden while providing a clear output. It also encourages immutability because the original data isn't changed unless the returned value is explicitly assigned. Functions that return values tend to be more reusable since their output can be used in many different contexts. Additionally, return values help with error handling by allowing functions to indicate if something went wrong. Capturing these values in variables improves code clarity, making the flow of data easier to follow. Using return values also promotes well-structured code, since each function has a clear input and output. Overall, this approach leads to programs that are modular, easier to maintain, and less prone to bugs.

```
1 def add_numbers(a, b): 1 usage
2     return a + b # Return the sum of a and b
3
4 result = add_numbers(a: 5, b: 7) # Capture the returned value in a variable
5 print(f"The result is {result}")
```

Figure 9: Shows how a function returns a value stored in a variable (result) for later use.

## Classes and Functions

Classes are used to group related data and functions into a single structure, making code easier to organize and reuse. A class works like a blueprint for creating objects, which are separate copies that each hold their own values. Functions written inside a class are called methods and are used to perform actions or return results based on the data stored in the object. Keeping data and functions together in this way helps with code organization and supports encapsulation by limiting what parts of the program can access or change specific values. Static methods can be added to a class when a task is related to the class but doesn't need to use any of the object's data. These are helpful for utility functions that still belong with the class logically. Classes and functions can also include document strings, or docstrings, which describe what the code does and how it should be used. Including these descriptions helps others understand the code more easily and supports better documentation. Writing programs with classes and functions leads to cleaner structure, easier debugging, and more readable code (Figure 10). This makes larger programs easier to manage and update over time.

```
1 class Student: 2 usages
2     """Represents a student with a name and GPA."""
3
4     def __init__(self, name, gpa):
5         self.name = name
6         self.gpa = gpa
7
8     def display_info(self): 1 usage
9         """Prints the student's name and GPA."""
10        print(f"{self.name} has a GPA of {self.gpa}.")
11
12    @staticmethod 1 usage
13    def school_name():
14        """Prints the name of the school."""
15        print("Seattle High School")
16
17    # Create an instance of the Student class
18    student1 = Student(name="Alex", gpa=3.8)
19    student1.display_info()
20
21    # Call the static method
22    Student.school_name()
```

Figure 10: Shows the use of classes *Student*, the static method for *school\_name*, and descriptive docstrings.

## The Separation of Concerns (SoC) Pattern

The Separation of Concerns is a design principle that organizes code into clear sections, each with a specific role in the program. It helps to break down complex tasks into manageable parts, making the code easier to understand and maintain. The three main concerns in most programs are presentation, logic, and data storage. Presentation handles user interaction, such as displaying messages or collecting input through forms or menus. Logic focuses on how the program processes information, including calculations and decision-making. Data storage manages how information is saved and retrieved, whether from files, databases, or other sources. These concerns are divided into layers: the presentation layer, the processing layer, and the data layer (Figure 11). Each layer interacts with the others in a controlled way, passing information without overlapping responsibilities. This structure promotes modularity by allowing each part to be worked on independently. It also improves scalability by making it easier to add new features without affecting unrelated sections. Code becomes more reusable because functions and modules are focused on specific tasks. Maintainability improves as developers can locate and fix problems more quickly. Documenting each layer and using clear boundaries between concerns helps with collaboration in group projects. The Separation of Concerns allows for cleaner, more organized programs that are easier to build and support.

```
1  # --- Data Layer ---
2  def get_discount_rate(): 1 usage
3      return 0.1 # 10% discount
4
5  # --- Processing Layer ---
6  def calculate_discounted_price(price): 1 usage
7      discount = get_discount_rate()
8      return price - (price * discount)
9
10 # --- Presentation Layer ---
11 def main(): 1 usage
12     print("Discount Calculator")
13     try:
14         price = float(input("Enter the original price: $"))
15         final_price = calculate_discounted_price(price)
16         print(f"Price after discount: ${final_price:.2f}")
17     except ValueError:
18         print("Please enter a valid number.")
19
20 # Run the program
21 main()
```

Figure 11: Shows the organization of the Data Layer, Processing Layer, and Presentation Layer.

## The Assignment

To redesign the course registration application using the concepts introduced in Module 06, the program was restructured around functions, classes, and the Separation of Concerns programming pattern. The script begins by defining two constants: one for the file name (Enrollments.json) and another for the menu text that is shown each time the user is prompted (Figure 12). These constants are declared at the top of the file to clearly separate values that should remain unchanged throughout execution. The program's global list, students, stores all student registration records as dictionaries containing three key-value pairs: "FirstName", "LastName", and "CourseName". The use of dictionaries improves data clarity and structure, while JSON is used for persistent storage to maintain data between sessions (Figure13).

```
11 #Data Constants
12 MENU: str = '''
13 ---- Course Registration Program ----
14     Select from the following menu:
15     1. Register a Student for a Course
16     2. Show current data
17     3. Save data to a file
18     4. Exit the program
19     -----
20     '''
21 FILE_NAME: str = "Enrollments.json"
```

Figure 12:

```
100 #Main Body
101 students: list = []
102 students = FileProcessor.read_data_from_file(FILE_NAME, students)
```

Figure 13:

The program was reorganized into two classes: FileProcessor (Figure 14) and IO (Figure 15.1 & 15.2), each containing static methods that handle specific tasks. The FileProcessor class manages all data file operations, including reading and writing the JSON file. The IO class handles user-facing tasks like displaying the menu, collecting input, and showing output. This structure reflects the Separation of Concerns principle, in which each class has a distinct responsibility. At startup, the script calls a method from FileProcessor to load any existing registration records from Enrollments.json, so that returning users can continue where they left off. Error handling using try/except ensures the program handles missing or unreadable files without crashing.



```

23 #Processing Classes
24 class FileProcessor: 2 usages
25     """
26     Handles reading and writing student data to a file.
27     """
28
29     @staticmethod 1 usage
30     def read_data_from_file(file_name: str, student_data: list):
31         try:
32             with open(file_name, "r") as file:
33                 student_data.extend(json.load(file))
34         except FileNotFoundError as e:
35             IO.output_error_messages(message: "File not found.", e)
36         except Exception as e:
37             IO.output_error_messages(message: "General error reading file.", e)
38         return student_data
39
40     @staticmethod 1 usage
41     def write_data_to_file(file_name: str, student_data: list):
42         try:
43             with open(file_name, "w") as file:
44                 json.dump(student_data, file)
45                 print("The following data was saved to file!")
46                 IO.output_student_courses(student_data)
47         except Exception as e:
48             IO.output_error_messages(message: "Error saving to file.", e)

```

Figure 14:

```

51 #Presentation (Input/Output) Classes
52 class IO: 10 usages
53     """
54     Handles input and output from the user.
55     """
56
57     @staticmethod 5 usages
58     def output_error_messages(message: str, error: Exception = None):
59         print(f"Error: {message}")
60         if error:
61             print("-- Technical Error Message --")
62             print(error.__doc__)
63             print(error.__str__())
64
65     @staticmethod 1 usage
66     def output_menu(menu: str):
67         print(menu)
68
69     @staticmethod 1 usage
70     def input_menu_choice():
71         return input("What would you like to do: ")
72
73     @staticmethod 2 usages
74     def output_student_courses(student_data: list):
75         print("-" * 50)
76         for student in student_data:
77             print(f'Student {student["FirstName"]} {student["LastName"]} is enrolled in {student["CourseName"]}')
78         print("-" * 50)
79
80     @staticmethod 1 usage
81     def input_student_data(student_data: list):

```

Figure 15.1:

```
80     @staticmethod 1 usage
81     def input_student_data(student_data: list):
82         try:
83             first_name = input("Enter the student's first name: ")
84             if not first_name.isalpha():
85                 raise ValueError("The first name should not contain numbers.")
86             last_name = input("Enter the student's last name: ")
87             if not last_name.isalpha():
88                 raise ValueError("The last name should not contain numbers.")
89             course_name = input("Please enter the name of the course: ")
90             student = {"FirstName": first_name, "LastName": last_name, "CourseName": course_name}
91             student_data.append(student)
92             print(f"You have registered {first_name} {last_name} for {course_name}.")
93         except ValueError as e:
94             IO.output_error_messages(str(e), e)
95         except Exception as e:
96             IO.output_error_messages(message="General input error.", e)
97         return student_data
```

Figure 15.2:

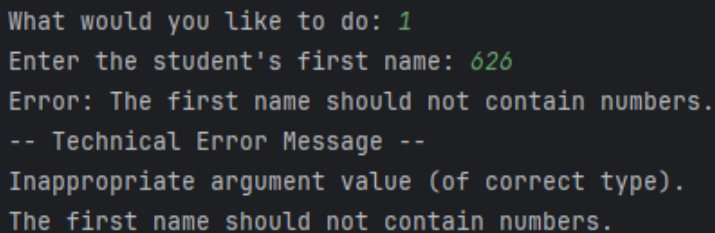
The menu-driven logic remains inside a while loop, but now each menu action now calls a specific function from the IO or FileProcessor class (Figure 16). When the user chooses option 1, the `input_student_data` method prompts for a student's first name, last name, and course name. Input validation is included to ensure names do not contain numbers. A new dictionary is created with the input values and appended to the students list. Selecting option 2 calls `output_student_courses`, which prints each student's registration as a clear, descriptive sentence. With the data structured as dictionaries, it becomes easier to access each field by name rather than by position, reducing the chance of logic errors.

```
104     while True:
105         IO.output_menu(MENU)
106         menu_choice = IO.input_menu_choice()
107
108         if menu_choice == "1":
109             students = IO.input_student_data(students)
110         elif menu_choice == "2":
111             IO.output_student_courses(students)
112         elif menu_choice == "3":
113             FileProcessor.write_data_to_file(FILE_NAME, students)
114         elif menu_choice == "4":
115             break
116         else:
117             print("Please only choose option 1, 2, 3, or 4")
118
119     print("Program Ended")
```

Figure 16:

Selecting option 3 writes all current registrations to the JSON file using `json.dump`. The file is overwritten with the updated list, and the output is shown to confirm the save. If an error

occurs during this process such as an invalid data type or a file access issue then a custom error message is shown without ending the program (Figure 17). These improvements result in a more organized, maintainable application that is easier to troubleshoot and extend. The final JSON file is formatted and easy to read containing each student's record stored as a clearly defined dictionary within a list, mirroring the structure used in the program's logic.

A terminal window with a dark background and light green text. The text shows a sequence of prompts and user input, followed by an error message. The prompts are 'What would you like to do: ', 'Enter the student's first name: ', and 'Error: The first name should not contain numbers.'. The user input is '1' and '626'. The error message is followed by a separator line '-- Technical Error Message --' and a detailed error description: 'Inappropriate argument value (of correct type). The first name should not contain numbers.'

```
What would you like to do: 1
Enter the student's first name: 626
Error: The first name should not contain numbers.
-- Technical Error Message --
Inappropriate argument value (of correct type).
The first name should not contain numbers.
```

Figure 17:

## Summary

By applying structured programming habits, developers are able to write code that is both functional and well-organized. Making consistent use of local variables, parameterized functions, and return values helps to isolate logic and minimize dependencies between components. Classes provide a clean way to bundle related data and behaviors together, which enhances clarity and promotes reuse across different parts of a program. The Separation of Concerns pattern strengthens this organization by clearly dividing responsibilities between presentation, processing, and data management layers. Each tool contributes to professional coding practices by encouraging encapsulation and reducing complexity. Modular design not only improves readability but also simplifies testing and debugging, especially as projects grow larger. As more features are added, structured code is easier to scale and maintain without introducing unnecessary errors. This approach also supports collaboration by making it easier for multiple developers to work on different parts of the same program. When each section of code is built with a specific role and clear boundaries, the result is a more robust and efficient application. Developing the habit of using these concepts early lays the foundation for writing high-quality code in any programming environment.