

Mariam Assaad

Wednesday, November 27, 2024

IT FDN 100 A - Foundations of Programming: Python

Assignment 07

[GitHub URL](#)

Expanding Python Skills: Exploring Classes, Constructors, and Inheritance

Introduction

This document explores fundamental and advanced concepts in Python programming, focusing on object-oriented programming (OOP) principles and their practical applications. It highlights the importance of structuring code effectively using statements, functions, classes, constructors, properties, and inheritance. Each section builds on the previous one, showcasing how these components contribute to creating modular, maintainable, and scalable programs.

The progression begins with understanding how statements are organized into reusable functions, and functions into classes, laying the groundwork for efficient code organization. It continues with constructors, which ensure that objects are created with a defined state, and properties, which provide an efficient way to manage and validate attributes. Finally, the document delves into inheritance, a powerful feature of OOP that promotes reusability and abstraction by enabling child classes to build upon the functionality of parent classes. This document provides a comprehensive guide to mastering these concepts.

1. Classes and Functions

In this section, I learned about the three fundamental components of Python programming: statements, functions, and classes, and how they come together to create structured and maintainable code. Statements, such as data assignments, loops, and conditionals, form the basic building blocks of any program. Organizing these statements into functions allows repetitive tasks to be simplified and reused, making the code more efficient and easier to debug.

I also explored how functions can be grouped into classes for better organization, especially in larger projects. This approach makes the code modular and scalable, as classes can serve as templates for creating multiple independent objects. For example, creating objects from a class like `Person` allows each instance to have its own unique data while following the same structure.

The concept of data encapsulation stood out, showing how objects maintain their own memory space, keeping data isolated and unaffected by other objects. I also learned about the distinction between data-focused classes, like `Person`, and processing-focused classes, such as `FileProcessor`, which helps separate responsibilities in the code. Overall, this section highlighted how using statements, functions, and classes together can transform a simple script into a well-structured program that is easier to manage and expand over time.

```
class Person:
    """A class representing a person"""

    def __init__(self, first_name: str = "", last_name: str = ""):
        self.first_name = first_name
        self.last_name = last_name

    def __str__(self):
        return f"{self.first_name} {self.last_name}"

class Student(Person):
    """A class representing a student"""

    def __init__(self, first_name: str = "", last_name: str = "",
course_name: str = ""):
        super().__init__(first_name, last_name)
        self.course_name = course_name

    def __str__(self):
        return f"{super().__str__()} registered for {self.course_name}"
```

Figure 1: Defining the Person and Student Classes

2. Using Constructors

In this module section, I gained a deeper understanding of constructors in Python and their role in initializing an object's attributes when the object is created. A constructor, defined with the `__init__` method, ensures that an object starts with a well-defined state by assigning initial values to its attributes. This is particularly useful for organizing data efficiently and enforcing data integrity at the time of object creation.

One key takeaway was the ability to simplify code by defining instance variables directly within the constructor, rather than as class-level attributes. This approach allows for cleaner, more concise code. Additionally, the `self` keyword plays a central role in constructors, referring to the specific instance of the class being operated on, which ensures that each object maintains its own independent data.

The module also highlighted best practices for implementing data validation within constructors to ensure that only valid data is assigned to attributes. For example, methods can include checks to verify that certain conditions are met before setting attribute values. Furthermore, the concept of private attributes was introduced, where attributes are marked as private by using double underscores (`__`), making them inaccessible from outside the class. This promotes encapsulation and prevents unintended modifications.

Through this section, I also explored how constructors facilitate object-oriented programming (OOP) by enabling the creation of multiple object instances from the same class, each with its own unique attributes. Overall, the module emphasized the importance of constructors in creating robust and well-structured programs.

```
class Student(Person):
    def __init__(self, first_name: str = "", last_name: str = "",
course_name: str = ""):
        super().__init__(first_name, last_name)
        self.course_name = course_name
```

Figure 2: Constructor in the Student Class

Figure 2 demonstrates how constructors in the child class (Student) utilize the `super()` function to call the parent class's constructor (Person) while adding additional initialization.

3. Using Properties

In this section, I delved into the use of properties in Python, an advanced concept that streamlines how attributes are accessed and validated in object-oriented programming. Properties, implemented using `@property` for getters and `@property_name.setter` for setters, allow for cleaner and more intuitive code while maintaining robust data validation.

The key insight was how properties replace traditional “get” and “set” methods, offering a more Pythonic way to encapsulate data within a class. This approach ensures that validation logic is consistently applied whenever an attribute's value is accessed or modified. For example, the getter property can format data (e.g., title-casing names), and the setter can enforce constraints (e.g., disallowing numeric characters in a name).

This section also emphasized abstraction and encapsulation. By managing private attributes with properties, internal logic is hidden from external code, presenting a simplified interface for interacting with objects. This not only enhances code readability but also reduces the risk of accidental data corruption.

Additionally, properties were seamlessly integrated into constructors, ensuring that validation is applied during object creation as well as when attributes are updated later. This dual functionality demonstrates how properties support the principles of object-oriented programming while maintaining flexibility and reliability in software design.

The following code, figure 3, illustrates the use of properties to encapsulate and validate data for `first_name`, `last_name`, and `course_name`. Properties allow robust data validation and ensure attributes meet specific criteria.

```
@property
def first_name(self):
    return self.__first_name.title()

@first_name.setter
def first_name(self, value: str):
    if value.isalpha() or value == "":
        self.__first_name = value
    else:
        raise ValueError("First name must contain only alphabetic characters.")

@property
def last_name(self):
    return self.__last_name.title()

@last_name.setter
def last_name(self, value: str):
    if value.isalpha() or value == "":
        self.__last_name = value
    else:
        raise ValueError("Last name must contain only alphabetic characters.")
```

Figure 3: Properties in the Person Class

4. Using Inheritance

This section introduced inheritance, a fundamental concept in object-oriented programming (OOP), which enables a child class to inherit properties and behaviors from a parent class. This approach simplifies code by reusing existing functionality while allowing extensions or modifications in the child class. For instance, Python inherently uses a root class named `Object`, which provides base-level methods shared by all classes.

Key insights from this section include the use of magic methods like `__init__` for constructors and `__str__` for creating string representations of objects. These methods enhance usability by offering default functionality that developers can customize. For example, while the

default `__str__` method displays an object's memory address, overriding it enables a more user-friendly output, such as displaying a student's name and GPA.

The concept of method overriding was particularly impactful. It involves redefining inherited methods in child classes to customize their behavior while retaining the core functionality of the parent class. This principle was demonstrated by creating a Student class that inherits attributes like `first_name` and `last_name` from a Person parent class but adds its unique attribute, GPA.

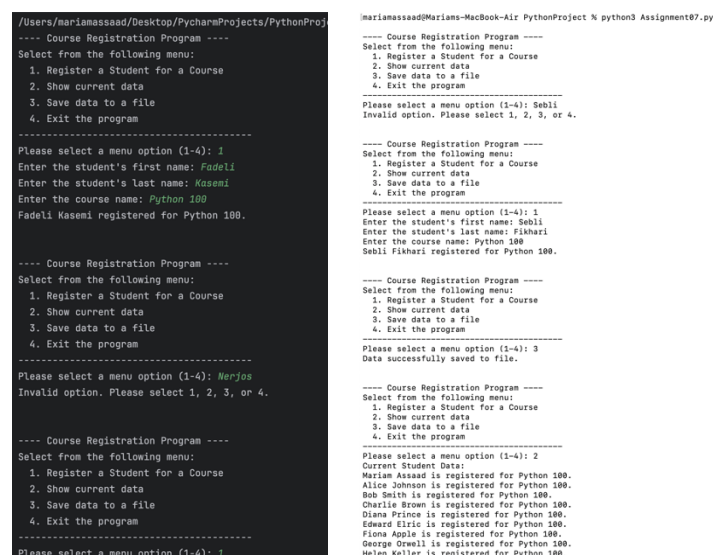
Below, figure 4, demonstrates inheritance by showing how the Student class inherits attributes and methods from the Person class. It also illustrates method overriding by redefining the `__str__` method in the Student class.

```
class Student(Person):
    def __str__(self):
        return f"{super().__str__() } registered for {self.course_name}"
```

Figure 4: Inheritance and Overriding the `__str__` Method

The advantages of inheritance were highlighted in scenarios involving multiple related classes, such as Teacher, Student, and Staff, where shared attributes are centralized in the parent class. This not only promotes consistency but also reduces code redundancy. Overall, this section emphasized how inheritance enhances code maintainability and scalability by encouraging reusability and abstraction.

Testing and Running Assignment 6



```
----- 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

Please select a menu option (1-4): 1
Enter the student's first name: Fadeli
Enter the student's last name: Kasemi
Enter the course name: Python 100
Fadeli Kasemi registered for Python 100.

----- 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

Please select a menu option (1-4): 2
Invalid option. Please select 1, 2, 3, or 4.

----- 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

Please select a menu option (1-4): 3
Data successfully saved to file.

----- 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

Please select a menu option (1-4): 2
Current Student Data:
Marian Assad is registered for Python 100.
Alice Johnson is registered for Python 100.
Bob Smith is registered for Python 100.
Charlie Brown is registered for Python 100.
Diana Prince is registered for Python 100.
Edward Elric is registered for Python 100.
Fiona Apple is registered for Python 100.
George Orwell is registered for Python 100.
Helen Keller is registered for Python 100.
```

Figure 5: The program runs correctly in both PyCharm and from the console or terminal.

Summary

This document provided an in-depth exploration of Python's object-oriented programming (OOP) principles and their application in creating structured and maintainable code. It began with the foundational concepts of statements, functions, and classes, demonstrating how these components work together to form the backbone of any program. The journey progressed through the use of constructors to initialize objects, properties to manage and validate attributes, and inheritance to promote reusability and scalability in code. Each section emphasized best practices, such as using constructors for data validation, properties for encapsulation and abstraction, and inheritance for reducing redundancy while extending functionality. By implementing these principles, programs become more modular, easier to understand, and adaptable to future changes. Overall, this document highlighted the importance of clean, efficient, and reusable code in Python programming. Mastering these concepts not only enhances coding efficiency but also lays a strong foundation for tackling more advanced programming challenges in the future. Through practical examples and clear explanations, this guide serves as a valuable resource for building robust Python applications.