Object-Oriented Programming - MAD 102 Week 9 Notes

Hia Al Saleh

October 30th, 2024

Contents

1 Introduction to Programming Paradigms

1.1 Procedural Programming

Procedural programming involves writing programs as a series of instructions to be executed sequentially. Common elements in procedural programming include:

- **Sequence**: Organizing code to be executed in a specific order.
- **Conditionals and Loops**: Adding control flow with if-else statements and loops.
- **Functions**: Grouping code into reusable blocks.

This approach is effective for small programs but presents challenges with complexity:

- Code becomes harder to manage as the program size increases.
- Variables are often global, leading to unexpected interactions.
- Does not directly associate data with the functions operating on it.

1.2 Object-Oriented Programming (OOP)

OOP overcomes these limitations by organizing code around **objects**, which represent real-world entities or concepts. Each object combines data and methods, making the code more modular, easier to understand, and maintainable. Key benefits include:

- **Data Integrity**: Attributes and behaviors are encapsulated within objects, preventing unauthorized access.
- **Reusability**: Allows code reuse through inheritance and polymorphism.
- **Extensibility**: New functionalities can be easily added by extending existing classes.

2 Core Concepts in OOP

2.1 Objects and Classes

In OOP, we use **classes** as blueprints to create **objects**. For instance, a `Car` class might define attributes like `color`, `make`, and `model`, and behaviors like `drive()` or `brake()`.

- **Objects** are instances of classes, containing data and methods.
- **Classes** define the structure and behavior of objects.

2.2 Attributes and Methods

Attributes store the state of an object (e.g., a dog's `breed`), while **methods** define its behavior (e.g., a dog's `bark()` method). Together, they form a self-contained entity.

```
class Dog:
    def __init__(self, breed):
        self.breed = breed

    def bark(self):
        print("Woof!")
```

2.3 Encapsulation

Encapsulation refers to restricting access to an object's internal data, allowing control over modifications. This improves security and simplifies code maintenance:

- **Public Attributes and Methods**: Accessible from outside the class.
- **Private Attributes and Methods**: Restricted to the class itself.

For example, `_balance` might be a private attribute in a `BankAccount` class, accessible only through methods like `deposit()` or `withdraw()`.

2.4 Abstraction

Abstraction hides complex details, exposing only relevant information. For example:

- Logging into an account shows a login form but hides the details of password encryption.

By focusing on essential information, abstraction reduces complexity for users.

2.5 Inheritance

Inheritance allows classes to share attributes and methods, simplifying code reuse:

- **Base (Parent) Class**: Defines general characteristics.
- **Derived (Child) Class**: Inherits from the base and adds specific features.

Example:

```
class Animal:
    def move(self):
        print("Moving...")

class Dog(Animal):
    def bark(self):
        print("Woof!")
```

Here, `Dog` inherits `move()` from `Animal` while adding `bark()`.

2.6 Polymorphism

Polymorphism enables classes to redefine inherited methods, allowing different behaviors based on the object's type. For instance:

```
class Animal:
    def make_noise(self):
        print("Some noise")

class Cat(Animal):
    def make_noise(self):
        print("Meow")

class Dog(Animal):
    def make_noise(self):
        print("Woof")
```

Calling `make_noise()` on a `Dog` or `Cat` object will produce different sounds, demonstrating polymorphism.

3 Creating and Using Classes

3.1 Defining Classes

A class defines the properties and methods for its objects. In Python, we define a class using the `class` keyword.

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

The `__init__()` method is a constructor that initializes a new instance of the class.

3.2 Creating Instances

Once a class is defined, we can create instances (objects) by calling the class as if it were a function.

```
student1 = Student("Alice", 123)
student2 = Student("Bob", 456)
```

3.3 Instance and Class Attributes

- **Instance Attributes**: Specific to each object, like `student_id`.
- **Class Attributes**: Shared across all instances, defined within the class scope.

Example:

```
class School:
    school_name = "Greenwood High"  # Class attribute

    def __init__(self, name):
        self.name = name  # Instance attribute
```

4 Memory Management in OOP

Memory management in OOP involves handling memory allocation and deallocation. In Python:

- **Automatic Memory Management**: Python manages memory allocation.
- **Garbage Collection**: When an object's reference count reaches zero, it's deallocated.

5 Summary of Key Concepts

OOP offers a structured way to create complex programs by grouping related data and functionality. The main pillars—**Encapsulation**, **Abstraction**, **Inheritance**, and **Polymorphism**—provide a foundation for building modular, reusable, and scalable code.