# UNIT IV: OOPs in Python

## 1. Classes in Python

### 1.1 Introduction to Classes

A **class** in Python is a blueprint for creating objects. Objects are instances of classes and can have attributes (variables) and methods (functions). Classes allow you to bundle data and functionality together.

#### Syntax for Creating a Class:

```
class ClassName:
    # Class body
    pass
```

### 1.2 Features of Classes

- **Encapsulation**: Classes encapsulate data (attributes) and functions (methods) that manipulate the data.
- **Reusability**: Once a class is defined, you can create multiple instances (objects) from it.
- **Organization**: Classes help in organizing and structuring the code effectively.

#### Example

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

    def speak(self):
        print(f"{self.name} says woof!")

# Creating an object of Dog class
dog1 = Dog("Buddy", 3)
dog1.speak()  # Output: Buddy says woof!
```

---

## 2. Principles of Object Orientation

Object-Oriented Programming (OOP) is based on several core principles that help manage complexity and improve code organization.

## 2.1 The Four Main Principles:

1. **Encapsulation**: Bundling the data (attributes) and methods that operate on the data within one unit (class). It restricts access to some of the object's components, making it easier to maintain.

   Example:

```
class Car:
    def __init__(self, brand, model):
        self.__brand = brand  # Private variable
        self.model = model

    def get_brand(self):
        return self.__brand

car1 = Car("Toyota", "Corolla")
print(car1.get_brand())  # Output: Toyota
```

2. **Abstraction**: Hiding complex implementation details and showing only the necessary features of an object. This allows users to interact with the object at a higher level of abstraction.

   Example:

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        print("Woof")

dog = Dog()
dog.speak()  # Output: Woof
```

3. **Inheritance**: A mechanism that allows one class to inherit the attributes and methods of another class, promoting reusability and creating a hierarchical class structure.

   Example:

```
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Woof")

dog = Dog()
dog.speak()  # Output: Woof
```

4. **Polymorphism**: The ability of an object to take on many forms. It allows you to use a single interface for different data types, enabling code reusability.

   Example:

```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        print("Woof")

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

animals = [Dog(), Cat()]
for animal in animals:
    animal.speak()  # Output: Woof Meow
```

---

# 3. Creating Classes in Python

## 3.1 The `__init__()` Method

The `__init__()` method is a special method called a **constructor**, which initializes the attributes of an object when an instance of the class is created.

This method is called automatically when an object is created from a class, and it initializes the attributes of the object.

**Syntax:**

```
class ClassName:
    def __init__(self, parameter1, parameter2):
        self.attribute1 = parameter1
        self.attribute2 = parameter2
```

**Example:**

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

# Creating an object
car1 = Car("Toyota", "Corolla")
print(car1.brand)  # Output: Toyota
```

---

# 4. Instance Methods

**Instance methods** are functions that are defined inside a class and are used to modify or interact with the instance attributes. These methods operate on individual objects of the class and can access their specific data.

- Instance methods are functions that operate on an instance of the class. These methods can access and modify the instance's attributes.

The first parameter of any instance method is `self`, which refers to the current instance of the class.

**Example:**

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

rect = Rectangle(4, 5)
print(rect.area())  # Output: 20
```

---

# 5. Class Variables

**Class variables** are variables that are shared by all instances of a class. They are defined inside the class but outside of any methods. Class variables are used when you need to store data that is common to all instances of the class.

Class variables are not tied to a particular object, but to the class itself.

- Class variables are variables that are shared by all instances of the class. They are defined outside of any methods, typically at the top of the class.

**Example:**

```
class Dog:
    species = "Canis familiaris"  # Class variable

    def __init__(self, name, age):
        self.name = name  # Instance variable
        self.age = age

dog1 = Dog("Buddy", 3)
dog2 = Dog("Charlie", 5)
print(dog1.species)  # Output: Canis familiaris
print(dog2.species)  # Output: Canis familiaris
```

# 6. Inheritance

Inheritance allows one class to inherit the attributes and methods of another class. The class being inherited from is called the **base class** or **parent class**, and the class inheriting is called the **derived class** or **child class**.

**Types of Inheritance:**

- **Single Inheritance**: A class inherits from one parent class.
- **Multiple Inheritance**: A class inherits from more than one parent class.
- **Multilevel Inheritance**: A class inherits from a child class, forming a hierarchy.
- **Hierarchical Inheritance**: Multiple classes inherit from a single parent class.

**Example:**

```
class Animal:
    def sound(self):
        print("Some sound")

class Dog(Animal):
    def sound(self):
        print("Bark")

dog = Dog()
dog.sound()  # Output: Bark
```

---

## 7. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common base class. It also allows methods in derived classes to have the same name but behave differently.

- In Simple terms, Polymorphism in Python allows different objects to respond to the same method in their own way. This makes code more flexible and extensible. Polymorphism allows objects of different classes to be treated as objects of a common base class, making it easier to scale and maintain large codebases.

**Example:**

```
class Bird:
    def speak(self):
        print("Chirp")

class Dog:
    def speak(self):
        print("Bark")

animals = [Bird(), Dog()]
for animal in animals:
    animal.speak()  # Output: Chirp Bark
```

---

## 8. Type Identification

You can use Python's `type()` function to determine the type of an object.

This is useful when you want to check if an object is of a particular type before performing an operation on it.

**Example:**

```
x = 5
print(type(x))  # Output: <class 'int'>

y = "Hello"
print(type(y))  # Output: <class 'str'>
```

## 8.2 `isinstance()` Function

`isinstance()` is used to check if an object is an instance of a particular class or a subclass of that class.

**Example:**

```
x = 5
print(isinstance(x, int))  # Output: True
```