Python OOPs Concepts

A class in Python is a user-defined template for creating objects. It bundles data and functions together, making it easier to manage and use them. When we create a new class, we define a new type of object. We can then create multiple instances of this object type.

Classes are created using class keyword. Attributes are variables defined inside the class and represent the properties of the class. Attributes can be accessed using the dot . operator (e.g., MyClass.my_attribute).

In [1]:
```
# define a class
class Dog:
    sound = "bark"  # class attribute
```

Create Object
An Object is an instance of a Class. It represents a specific implementation of the class and holds its own data.

Now, let's create an object from Dog class.

In [3]:
```
class Dog:
    sound = "bark"

# Create an object from the class
dog1 = Dog()

# Access the class attribute
print(dog1.sound)


#sound attribute is a class attribute. It is shared across all instances of Dog c
#so can be directly accessed through instance dog1.
```

```
bark
```

In [4]:
```
#Using __init__() Function
#In Python, class has __init__() function. It automatically initializes object at

class Dog:
    species = "Canine"  # Class attribute

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

#Explanation:

#class Dog: Defines a class named Dog.
#species: A class attribute shared by all instances of the class.
#__init__ method: Initializes the name and age attributes when a new object is cr
```

In [5]:
```
class Dog:
    species = "Canine"  # Class attribute

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

# Creating an object of the Dog class
dog1 = Dog("Buddy", 3)

print(dog1.name)  # Output: Buddy
print(dog1.species)  # Output: Canine
```

```
Buddy
Canine
```

In [6]:
```
#Self Parameter
#self parameter is a reference to the current instance of the class. It allows us
#of the object.

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

    def bark(self):
        print(f"{self.name} is barking!")

# Creating an instance of Dog
dog1 = Dog("Buddy", 3)
dog1.bark()

#Explanation:
#Inside bark(), self.name accesses the specific dog's name and prints it.
#When we call dog1.bark(), Python automatically passes dog1 as self, allowing acc
```

```
Buddy is barking!
```

In [7]:
```
#__str__ Method
#__str__ method in Python allows us to define a custom string representation of c
#By default, when we print an object or convert it to a string using str(),
#Python uses the default implementation, which returns a string like <__main__.Cl


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

    def __str__(self):
        return f"{self.name} is {self.age} years old."  # Correct: Returning a st

dog1 = Dog("Buddy", 3)
dog2 = Dog("Charlie", 5)

print(dog1)
print(dog2)
```

```
Buddy is 3 years old.
Charlie is 5 years old.
```

Explanation:

__str__ Implementation: Defined as a method in the Dog class. Uses the self parameter to access the instance's attributes (name and age).
Readable Output: When print(dog1) is called, Python automatically uses the __str__ method to get a string representation of the object. Without __str__, calling print(dog1) would produce something like <__main__.Dog object at 0x00000123>.

Class and Instance Variables in Python
In Python, variables defined in a class can be either class variables or instance variables, and understanding the distinction between them is crucial for object-oriented programming.

Class Variables
These are the variables that are shared across all instances of a class. It is defined at the class level, outside any methods. All objects of the class share the same value for a class variable unless explicitly overridden in an object.

Instance Variables
Variables that are unique to each instance (object) of a class. These are defined within __init__ method or other instance methods. Each object maintains its own copy of instance variables, independent of other objects.

Explanation:
Class Variable (species): Shared by all instances of the class. Changing Dog.species affects all objects, as it's a property of the class itself.
Instance Variables (name, age): Defined in the __init__ method. Unique to each instance (e.g., dog1.name and dog2.name are different).

```
Accessing Variables: Class variables can be accessed via the class name
(Dog.species) or an object (dog1.species). Instance variables are accessed via
the object (dog1.name).
Updating Variables: Changing Dog.species affects all instances. Changing
dog1.name only affects dog1 and does not impact dog2.
```

Polymorphism in Python

```
Polymorphism in Python allows different classes to be treated as instances of
the same class through a shared interface. This concept enables methods in
different classes to have the same name but behave differently based on the
class instance.


Explanation of Polymorphism Concepts Used

Method Overriding (Runtime Polymorphism):

The speak() method is defined in the Animal class.
The Dog and Cat classes override the speak() method.
The function make_animal_speak(animal) accepts different objects and calls
speak(), showcasing polymorphism.


Method Overloading (Compile-time Polymorphism):
Python does not support method overloading natively, but it can be simulated
using default arguments.
In the Calculator class, add() accepts 2 or 3 parameters, with c defaulting to
0.


Operator Overloading:
The Vector class overloads the + operator using the __add__ method.
Two vector objects can be added using v1 + v2, returning a new Vector object.
```

In [8]:
```
# Demonstration of Polymorphism in Python

# 1. Method Overriding (Runtime Polymorphism)
class Animal:
    def speak(self):
        return "Animal makes a sound"

class Dog(Animal):
    def speak(self):
        return "Dog barks"

class Cat(Animal):
    def speak(self):
        return "Cat meows"

# Function demonstrating polymorphism
def make_animal_speak(animal):
    print(animal.speak())

# 2. Method Overloading (Achieved using Default Arguments)
class Calculator:
    def add(self, a, b, c=0):  # Method overloading using default argument
        return a + b + c

# 3. Operator Overloading
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        """Overloading the + operator"""
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

# Main execution block
if __name__ == "__main__":
    # Method Overriding Example
    print("Method Overriding Demonstration:")
    animals = [Dog(), Cat(), Animal()]
    for animal in animals:
        make_animal_speak(animal)

    # Method Overloading Example
    print("\nMethod Overloading Demonstration:")
    calc = Calculator()
    print(f"Sum of 2 and 3: {calc.add(2, 3)}")
    print(f"Sum of 2, 3, and 5: {calc.add(2, 3, 5)}")

    # Operator Overloading Example
    print("\nOperator Overloading Demonstration:")
    v1 = Vector(2, 3)
    v2 = Vector(4, 5)
    v3 = v1 + v2  # Uses __add__ method
    print(f"Vector Addition: {v3}")
```

```
Method Overriding Demonstration:
Dog barks
Cat meows
Animal makes a sound

Method Overloading Demonstration:
Sum of 2 and 3: 5
Sum of 2, 3, and 5: 10

Operator Overloading Demonstration:
Vector Addition: Vector(6, 8)
```

Inheritance in Python

```
Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that
allows one class (child/derived class) to inherit attributes and methods from
another class (parent/base class). This promotes code reusability and reduces
redundancy.

Types of Inheritance in Python

Single Inheritance: One class inherits from another.
Multiple Inheritance: A class inherits from multiple parent classes.
Multilevel Inheritance: A class inherits from a child class of another parent
class.
Hierarchical Inheritance: Multiple child classes inherit from the same parent
class.
Hybrid Inheritance: A combination of multiple inheritance types.
```

In [9]:
```python
# Demonstration of Inheritance in Python

# 1. Single Inheritance.....................
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Animal makes a sound"

class Dog(Animal):  # Dog class inherits from Animal
    def speak(self):
        return f"{self.name} barks"

# 2. Multiple Inheritance.....................
class Parent1:
    def feature1(self):
        return "Feature 1 from Parent1"

class Parent2:
    def feature2(self):
        return "Feature 2 from Parent2"

class Child(Parent1, Parent2):  # Inheriting from both Parent1 and Parent2
    def child_feature(self):
        return "Child class has additional features"

# 3. Multilevel Inheritance.....................
class Grandparent:
    def grandparent_feature(self):
        return "Feature from Grandparent"

class Parent(Grandparent):  # Inheriting from Grandparent
    def parent_feature(self):
        return "Feature from Parent"

class ChildMultilevel(Parent):  # Inheriting from Parent
    def child_feature(self):
        return "Feature from Child"

# 4. Hierarchical Inheritance.....................
class Vehicle:
    def type(self):
        return "This is a vehicle"

class Car(Vehicle):  # Car inherits from Vehicle
    def wheels(self):
        return "Car has 4 wheels"

class Bike(Vehicle):  # Bike inherits from Vehicle
    def wheels(self):
        return "Bike has 2 wheels"

# 5. Hybrid Inheritance (Combining multiple types).....................
class Base:
    def base_feature(self):
        return "Feature from Base class"
```

```
class Derived1(Base):
    def derived1_feature(self):
        return "Feature from Derived1"

class Derived2(Base):
    def derived2_feature(self):
        return "Feature from Derived2"

class Hybrid(Derived1, Derived2):
    def hybrid_feature(self):
        return "Hybrid class combining Derived1 and Derived2"

# Main Execution
if __name__ == "__main__":
    # Single Inheritance
    print("Single Inheritance:")
    dog = Dog("Buddy")
    print(dog.speak())

    # Multiple Inheritance
    print("\nMultiple Inheritance:")
    child = Child()
    print(child.feature1())
    print(child.feature2())
    print(child.child_feature())

    # Multilevel Inheritance
    print("\nMultilevel Inheritance:")
    child_multilevel = ChildMultilevel()
    print(child_multilevel.grandparent_feature())
    print(child_multilevel.parent_feature())
    print(child_multilevel.child_feature())

    # Hierarchical Inheritance
    print("\nHierarchical Inheritance:")
    car = Car()
    bike = Bike()
    print(car.type(), "-", car.wheels())
    print(bike.type(), "-", bike.wheels())

    # Hybrid Inheritance
    print("\nHybrid Inheritance:")
    hybrid_obj = Hybrid()
    print(hybrid_obj.base_feature())
    print(hybrid_obj.derived1_feature())
    print(hybrid_obj.derived2_feature())
    print(hybrid_obj.hybrid_feature())
```

```
Single Inheritance:
Buddy barks

Multiple Inheritance:
Feature 1 from Parent1
Feature 2 from Parent2
Child class has additional features

Multilevel Inheritance:
Feature from Grandparent
Feature from Parent
Feature from Child

Hierarchical Inheritance:
This is a vehicle - Car has 4 wheels
This is a vehicle - Bike has 2 wheels

Hybrid Inheritance:
Feature from Base class
Feature from Derived1
Feature from Derived2
Hybrid class combining Derived1 and Derived2
```

Abstraction in Python

```
Abstraction is an Object-Oriented Programming (OOP) concept that hides
implementation details and only exposes essential features to the user. In
Python, abstraction is achieved using the ABC (Abstract Base Class) module.

Key Points of Abstraction in Python
Abstract Classes: Cannot be instantiated directly.
Abstract Methods: Defined in the abstract class but must be implemented by
subclasses.
Implemented Using ABC Module: The ABC class from abc module is used to create
abstract classes.
```

In [1]:
```python
# Demonstration of Abstraction in Python
from abc import ABC, abstractmethod

# Abstract Class
class Vehicle(ABC):
    def __init__(self, brand):
        self.brand = brand

    @abstractmethod
    def start(self):
        """Abstract method that must be implemented by child classes"""
        pass

    @abstractmethod
    def stop(self):
        pass

    def brand_name(self):
        """Concrete method (Not abstract, available for all child classes)"""
        return f"Brand: {self.brand}"

# Concrete Class 1 (Inherits from Vehicle)
class Car(Vehicle):
    def start(self):
        return f"{self.brand} Car is starting with a key"

    def stop(self):
        return f"{self.brand} Car is stopping by applying brakes"

# Concrete Class 2 (Inherits from Vehicle)
class Bike(Vehicle):
    def start(self):
        return f"{self.brand} Bike is starting with a self-start button"

    def stop(self):
        return f"{self.brand} Bike is stopping by pressing the brake lever"

# Main Execution
if __name__ == "__main__":
    # Uncommenting the below line will give an error since we cannot instantiate
    # vehicle = Vehicle("Generic")

    print("Abstraction Demonstration:")

    car = Car("Toyota")
    print(car.brand_name())
    print(car.start())
    print(car.stop())

    print("\n")

    bike = Bike("Honda")
    print(bike.brand_name())
    print(bike.start())
    print(bike.stop())
```

```
Abstraction Demonstration:
Brand: Toyota
Toyota Car is starting with a key
Toyota Car is stopping by applying brakes


Brand: Honda
Honda Bike is starting with a self-start button
Honda Bike is stopping by pressing the brake lever
```

Encapsulation in Python

```
Encapsulation is one of the fundamental principles of Object-Oriented
Programming (OOP) that restricts direct access to certain variables and
methods, ensuring controlled access to an object's data. This is achieved using
access modifiers:

Public (public_var): Accessible from anywhere.
Protected (_protected_var): Accessible within the class and subclasses.
Private (__private_var): Cannot be accessed directly outside the class but can
be accessed via getter and setter methods.
```

In [2]:
```python
# Demonstration of Encapsulation in Python

class BankAccount:
    def __init__(self, account_holder, balance):
        self.account_holder = account_holder  # Public attribute
        self._account_type = "Savings"  # Protected attribute
        self.__balance = balance  # Private attribute

    # Public Method
    def get_balance(self):
        return f"{self.account_holder}'s balance: ${self.__balance}"

    # Private Method (cannot be accessed directly)
    def __apply_interest(self):
        self.__balance *= 1.05  # Applying 5% interest

    # Public Method to access the private method
    def apply_interest_public(self):
        self.__apply_interest()
        return "Interest applied successfully!"

    # Getter Method for Private Attribute
    def get_private_balance(self):
        return self.__balance

    # Setter Method for Private Attribute (Controlled Access)
    def set_private_balance(self, amount):
        if amount >= 0:
            self.__balance = amount
        else:
            raise ValueError("Balance cannot be negative!")

# Main Execution
if __name__ == "__main__":
    print("Encapsulation Demonstration:")

    # Creating an Object
    account = BankAccount("Alice", 5000)

    # Accessing Public Attribute
    print(account.account_holder)  # Works fine

    # Accessing Protected Attribute (Allowed, but should be used carefully)
    print(account._account_type)

    # Accessing Private Attribute (Throws AttributeError)
    # print(account.__balance)  # Uncommenting this will raise an error

    # Accessing Private Attribute using Getter
    print(account.get_balance())

    # Trying to Modify Private Attribute Directly (Throws Error)
    # account.__balance = 10000  # Uncommenting this won't change the real privat

    # Using Setter Method to Modify Private Attribute
    account.set_private_balance(7000)
    print(account.get_balance())
```

```
    # Applying Interest using Public Method (Calls Private Method Internally)
    print(account.apply_interest_public())
    print(account.get_balance())
```

```
Encapsulation Demonstration:
Alice
Savings
Alice's balance: $5000
Alice's balance: $7000
Interest applied successfully!
Alice's balance: $7350.0
```

Exception Handling in Python

```
Exception handling is a crucial concept in programming that allows a program to
gracefully handle errors instead of crashing unexpectedly. In Python,
exceptions occur when an error is encountered during program execution, and
they can be handled using try, except, else, and finally blocks.


Why Use Exception Handling?
Prevents Program Crashes: Instead of stopping execution, Python handles the
error and allows the program to continue.
Improves Debugging: It helps identify and log errors instead of displaying
vague system errors.
Ensures Code Reliability: Allows graceful error handling, ensuring better user
experience.


Types of Python Exceptions
Python has several built-in exceptions, such as:

ZeroDivisionError: Division by zero.
TypeError: Operation on incompatible data types.
ValueError: Incorrect value given.
IndexError: Accessing an out-of-range index.
KeyError: Accessing a non-existent dictionary key.
FileNotFoundError: Trying to access a missing file.
IOError: Issues with input/output operations.
AttributeError: Trying to access an undefined attribute.
ImportError: Module import failure.
```

In [3]:
```python
# Demonstration of Exception Handling in Python

def divide_numbers(a, b):
    try:
        result = a / b
        return f"Result: {result}"
    except ZeroDivisionError:
        return "Error: Division by zero is not allowed!"
    except TypeError:
        return "Error: Invalid input type. Please provide numbers!"
    except Exception as e:  # Generic exception handling
        return f"Unexpected Error: {e}"
    finally:
        print("Execution of divide_numbers completed.")

def read_file(file_name):
    try:
        with open(file_name, 'r') as file:
            return file.read()
    except FileNotFoundError:
        return "Error: File not found!"
    except Exception as e:
        return f"Unexpected Error: {e}"

# Main Execution
if __name__ == "__main__":
    print("Exception Handling Demonstration:")

    # Handling ZeroDivisionError
    print(divide_numbers(10, 0))

    # Handling TypeError
    print(divide_numbers(10, "two"))

    # Handling a Valid Division
    print(divide_numbers(10, 2))

    # Handling FileNotFoundError
    print(read_file("non_existent_file.txt"))

    # Demonstrating try-except-else-finally
    try:
        num = int(input("Enter an integer: "))  # May raise ValueError
    except ValueError:
        print("Error: Invalid input! Please enter an integer.")
    else:
        print(f"You entered: {num}")
    finally:
        print("Execution of input handling completed.")
```

```
Exception Handling Demonstration:
Execution of divide_numbers completed.
Error: Division by zero is not allowed!
Execution of divide_numbers completed.
Error: Invalid input type. Please provide numbers!
Execution of divide_numbers completed.
Result: 5.0
Error: File not found!
Enter an integer: 2
You entered: 2
Execution of input handling completed.
```

In [ ]: