**Module 7: Classes In Python**

In Python, classes are a fundamental part of object-oriented programming (OOP). A class is like a blueprint for creating objects. Objects are instances of classes that can hold data (attributes) and have behaviors (methods).

**Syntax of a Class**

class ClassName:

# Constructor method

def \_\_init\_\_(self, parameter1, parameter2):

self.parameter1 = parameter1 # Attribute

self.parameter2 = parameter2

# Method

def display(self):

print(f"Parameter1: {self.parameter1}, Parameter2: {self.parameter2}")

**Example**

class Student:

def \_\_init\_\_(self, name, roll\_no):

self.name = name

self.roll\_no = roll\_no

def show\_details(self):

print(f"Name: {self.name}, Roll No: {self.roll\_no}")

# Create object

s1 = Student("Amit", 101)

s1.show\_details()

**Key Concepts**

| **Term** | **Description** |
| --- | --- |

|  |  |
| --- | --- |
| **Class---------------** | **Keyword to define a class** |

|  |  |
| --- | --- |
| **\_\_init\_\_ ------------()** | **Constructor method (called automatically when object is created)** |

|  |  |
| --- | --- |
| **self** | **------------------Refers to the current object instance** |

|  |  |
| --- | --- |
| **Object** | **--------------Instance of a class** |

|  |  |
| --- | --- |
| **Method** | **-------------Function defined inside a class** |
|  |  |

**Why Use Classes?**

* To organize code better
* To model real-world entities
* To implement reusable and maintainable code (OOP principles like encapsulation, inheritance, etc.)

**More Advanced Topics**

**1. Encapsulation**

Encapsulation means hiding internal details of how an object works and only exposing what’s necessary.

**✅Example:**

class Account:

def \_\_init\_\_(self, owner, balance):

self.owner = owner

self.\_\_balance = balance # Private attribute (name mangling)

def deposit(self, amount):

self.\_\_balance += amount

def get\_balance(self):

return self.\_\_balance

acc = Account("Amit", 1000)

acc.deposit(500)

print(acc.get\_balance()) # 1500

# print(acc.\_\_balance) ❌ Will raise an error

**2. Inheritance**

Inheritance allows one class (child) to inherit the properties and methods of another class (parent).

**✅Example:**

class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal): # Inheriting Animal

def bark(self):

print("Dog barks")

d = Dog()

d.speak()

d.bark()

**3. Polymorphism**

Polymorphism allows methods with the same name to behave differently depending on the object.

**✅ Example:**

class Cat:

def sound(self):

print("Meow")

class Dog:

def sound(self):

print("Bark")

# Common interface

def make\_sound(animal):

animal.sound()

make\_sound(Cat())

make\_sound(Dog())

**Exception Classes & Custom Exceptions**

**1. Built-in Exception Classes**

Python provides many built-in exception classes. Some common ones:

| Exception | Description |
| --- | --- |
| ValueError | Raised when a function gets invalid value |
| TypeError | Wrong data type |
| ZeroDivisionError | Division by zero |
| FileNotFoundError | File not found |
| KeyError | Missing dictionary key |

**2. Creating a Custom Exception**

You can create your own exception class by inheriting from Exception class.

✅ **Example:**

class InvalidAgeError(Exception):

"""Custom Exception for invalid age"""

pass

def check\_age(age):

if age < 18:

raise InvalidAgeError("Age must be 18 or older!")

else:

print("You are eligible to vote.")

try:

check\_age(15)

except InvalidAgeError as e:

print("Error:", e)