**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)

* **why we use classes and OOP** even if it looks long for small tasks

**Scalability** — Real Projects Are Big

Classes aren’t meant for one-line outputs — they’re for large applications with many related behaviors.

Example:

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def greet(self):

print(f"Hello, my name is {self.name} and I’m {self.age} years old.")

class Student(Person):

def study(self, subject):

print(f"{self.name} is studying {subject}.")

‘’’Now this makes sense — you can create *many* persons and students, each with their own data and behavior:’’’

p1 = Person("Amit", 35)

s1 = Student("Riya", 21)

p1.greet()

s1.greet()

s1.study("Python")

Once you’ve defined a class like Person, you can reuse it in multiple programs without rewriting logic — it becomes a **reusable component**.

Real life project “Bank Account Management System – GUI Version (Tkinter)”

# ----------------------------------------------

# Bank Account Management System (with Tkinter)

# ----------------------------------------------

import tkinter as tk

from tkinter import messagebox

# Custom Exception

class InsufficientBalanceError(Exception):

"""Raised when account balance is insufficient"""

pass

# Base Account Class

class Account:

def \_\_init\_\_(self, acc\_no, name, balance=0):

self.acc\_no = acc\_no

self.name = name

self.\_\_balance = balance

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

return f"₹{amount} deposited successfully!"

else:

return "Deposit amount must be positive!"

def withdraw(self, amount):

if amount <= 0:

return "Invalid withdrawal amount!"

elif amount > self.\_\_balance:

raise InsufficientBalanceError("Insufficient balance!")

else:

self.\_\_balance -= amount

return f"₹{amount} withdrawn successfully!"

def get\_balance(self):

return self.\_\_balance

def get\_details(self):

return f"Account No: {self.acc\_no}\nName: {self.name}\nBalance: ₹{self.\_\_balance}"

# GUI Application

class BankApp:

def \_\_init\_\_(self, root):

self.root = root

self.root.title("🏦 Bank Account Management System")

self.root.geometry("400x400")

self.root.config(bg="#f2f2f2")

self.account = None # will hold Account object

# --- UI Layout ---

tk.Label(root, text="Bank Account Management", font=("Arial", 16, "bold"), bg="#f2f2f2", fg="#003366").pack(pady=10)

# Account creation

tk.Label(root, text="Account No:", bg="#f2f2f2").pack()

self.acc\_no\_entry = tk.Entry(root)

self.acc\_no\_entry.pack()

tk.Label(root, text="Name:", bg="#f2f2f2").pack()

self.name\_entry = tk.Entry(root)

self.name\_entry.pack()

tk.Label(root, text="Initial Balance:", bg="#f2f2f2").pack()

self.balance\_entry = tk.Entry(root)

self.balance\_entry.pack()

tk.Button(root, text="Create Account", bg="#003366", fg="white", command=self.create\_account).pack(pady=5)

# Deposit and Withdraw

tk.Label(root, text="Amount:", bg="#f2f2f2").pack()

self.amount\_entry = tk.Entry(root)

self.amount\_entry.pack()

frame = tk.Frame(root, bg="#f2f2f2")

frame.pack(pady=5)

tk.Button(frame, text="Deposit", width=10, bg="green", fg="white", command=self.deposit\_amount).grid(row=0, column=0, padx=5)

tk.Button(frame, text="Withdraw", width=10, bg="orange", fg="white", command=self.withdraw\_amount).grid(row=0, column=1, padx=5)

# Display buttons

tk.Button(root, text="Check Balance", bg="#004c99", fg="white", command=self.show\_balance).pack(pady=5)

tk.Button(root, text="Account Details", bg="#004c99", fg="white", command=self.show\_details).pack(pady=5)

# --- Functional Methods ---

def create\_account(self):

try:

acc\_no = int(self.acc\_no\_entry.get())

name = self.name\_entry.get()

balance = float(self.balance\_entry.get())

if not name:

messagebox.showwarning("Warning", "Please enter a name.")

return

self.account = Account(acc\_no, name, balance)

messagebox.showinfo("Success", f"Account created successfully for {name}!")

except ValueError:

messagebox.showerror("Error", "Please enter valid numeric values for Account No and Balance.")

def deposit\_amount(self):

if self.account is None:

messagebox.showwarning("Warning", "Create an account first!")

return

try:

amount = float(self.amount\_entry.get())

msg = self.account.deposit(amount)

messagebox.showinfo("Deposit", msg)

except ValueError:

messagebox.showerror("Error", "Enter a valid amount!")

def withdraw\_amount(self):

if self.account is None:

messagebox.showwarning("Warning", "Create an account first!")

return

try:

amount = float(self.amount\_entry.get())

msg = self.account.withdraw(amount)

messagebox.showinfo("Withdraw", msg)

except ValueError:

messagebox.showerror("Error", "Enter a valid amount!")

except InsufficientBalanceError as e:

messagebox.showerror("Error", str(e))

def show\_balance(self):

if self.account:

balance = self.account.get\_balance()

messagebox.showinfo("Balance", f"Available balance: ₹{balance}")

else:

messagebox.showwarning("Warning", "No account found!")

def show\_details(self):

if self.account:

details = self.account.get\_details()

messagebox.showinfo("Account Details", details)

else:

messagebox.showwarning("Warning", "No account found!")

# Run the Application

if \_\_name\_\_ == "\_\_main\_\_":

root = tk.Tk()

app = BankApp(root)

root.mainloop()

You’ll see a simple window where you can:

* Create an account
* Deposit / Withdraw
* Check balance
* View account details