# Advanced Programing

python

## OOP

OOP (Object-Oriented Programming) in Python is a programming style based on the concept of objects — real-world entities that contain data (attributes) and behavior (methods).

## Topics & Types

| Concept | Meaning |
|---|---|
| **Encapsulation** | Hiding internal data; controlling access through methods |
| **Abstraction** | Hiding complex logic; exposing only necessary parts |
| **Inheritance** | Reusing code by deriving a new class from an existing one ( copying props) |
| **Polymorphism** | Different classes can use the same method name with different behavior |

## Class

A class is a blueprint for creating objects.
It defines:

- Attributes (data/variables)
- Methods (functions/behaviors)

But it does not hold real data — the object created from the class does.

## code

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

    def drive(self):
        print(f"{self.color} {self.brand} is driving")

# Creating objects from class
car1 = Car("Toyota", "Red")
car2 = Car("Honda", "Black")

car1.drive()  # Red Toyota is driving
car2.drive()  # Black Honda is driving
```

## Behind Code

| Term | Role |
|---|---|
| class Car | Blueprint for making car objects |
| __init__ | Constructor, runs when object is created |
| self | Refers to the current object |
| car1, car2 | Objects/instances of class |

## Encapsulation

Encapsulation is an OOP principle that hides internal data and allows access through controlled methods.
It protects the internal state of an object and ensures data security and integrity.

```
class BankAccount:
    def __init__(self, name, balance):
        self.name = name
        self.__balance = balance  # private variable

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def get_balance(self):  # getter method
        return self.__balance

    def set_balance(self, amount):  # setter method with check
        if amount >= 0:
            self.__balance = amount
        else:
            print("Invalid balance amount")

# Usage
acc = BankAccount("Quddus", 10000)
acc.deposit(5000)
print(acc.get_balance())   # ✅ 15000

acc.set_balance(-1000)     # ❌ Invalid balance amount
print(acc.get_balance())   # ✅ still 15000

# Direct access (not recommended)
# print(acc.__balance)     # ❌ AttributeError
ack")

car1.drive()  # Red Toyota is driving
car2.drive()  # Black Honda is driving
```

## Inheritance

Inheritance is an OOP concept where a child class (subclass) inherits attributes and methods from a parent class (superclass).
It helps with code reuse and extension.

```
class CNIC:
  def __init__(self,number,linkedAcc):
    self.Number = number
    self.AccoutNumber = linkedAcc
  def data(self):
    print(f'your data cnic {self.Number}  {self.AccoutNumber}')

class Bank(CNIC):
  def __init__(self, number, linkedAcc, Name, BankName, amount, qty):
    super().__init__(number, linkedAcc)
    self.Name = Name
    self.Bank = BankName
    self.Amount = amount
    self.Qty = qty

  def showBank(self):
    print(f"bank name {self.Bank} | acc: {self.Number} | linked: {"CNIC:
", self.AccoutNumber}")

  def ShowBuyDATA(self):
    totalamount = self.Amount * self.Qty
    print("Total Amount ", totalamount)

quddus = Bank("03118923","5412","quddus","NBF",12000,3)
quddus.data()
quddus.showBank()
quddus.ShowBuyDATA()
```

## Abstraction

Abstraction is an OOP principle that means hiding complex internal logic and showing only necessary details to the outside world.

```python
from abc import ABC, abstractmethod

# Abstract class
class BankAccount(ABC):
    def __init__(self, name, balance):
        self.name = name
        self.balance = balance

    @abstractmethod
    def deposit(self, amount):
        pass

    @abstractmethod
    def withdraw(self, amount):
        pass

# Concrete class
class SavingAccount(BankAccount):
    def deposit(self, amount):
        self.balance += amount
        print(f"{amount} deposited. New Balance: {self.balance}")

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print(f"{amount} withdrawn. New Balance: {self.balance}")
        else:
            print("Insufficient funds.")

# Usage
acc = SavingAccount("Quddus", 10000)
acc.deposit(2000)
acc.withdraw(3000)

acc = BankAccount("Ali", 5000)  # ❌ Error: Can't instantiate abstract class
```

## Import | Export Libs

Python allows you to import and export code (functions, classes, variables) between files/modules to keep your code organized and reusable.

**myFolder/math_utils.py**

```python
# math_utils.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

**myFolder/main.py | import full lib funs**

```python
# main.py
import math_utils

print(math_utils.add(5, 3))       # 8
print(math_utils.subtract(10, 4)) # 6
```

**myFolder/main.py | import specific lib funs**

```python
# main.py
from math_utils import add,substract

print(add(5, 3))       # 8
print(subtract(10, 4)) # 6
```