**Object-Oriented Programming (OOP) with Python**

Object-Oriented Programming (OOP) is a programming approach that structures code using **objects** that contain **data (attributes)** and **behavior (methods)**.

**1. What is a Class?**

A **class** is a blueprint for creating objects. It defines variables (attributes) and functions (methods) that the objects of the class will use.

**Components of a Class:**

* **Instance Variables** – Belong to a specific object, defined using self.variable\_name.
* **Methods** – Functions inside a class that operate on object data.
* **Class Variables** – Shared among all instances of a class.
* **Class Methods** – Operate on class variables using @classmethod.

**Example: Defining a Class**

class Car:

wheels = 4 # Class variable (shared by all instances)

def \_\_init\_\_(self, brand, model):

self.brand = brand # Instance variable

self.model = model # Instance variable

def display\_info(self):

return f"{self.brand} {self.model} with {Car.wheels} wheels"

# Creating objects

car1 = Car("Toyota", "Camry")

car2 = Car("Honda", "Civic")

print(car1.display\_info())

print(car2.display\_info())

**2. Designing Your Own Classes**

**Example: Bank Account Class**

class BankAccount:

bank\_name = "XYZ Bank" # Class variable

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

self.owner = owner # Instance variable

self.\_\_balance = balance # Private variable

def deposit(self, amount):

self.\_\_balance += amount

return f"Deposited {amount}. New balance: {self.\_\_balance}"

def withdraw(self, amount):

if amount <= self.\_\_balance:

self.\_\_balance -= amount

return f"Withdrew {amount}. Remaining balance: {self.\_\_balance}"

else:

return "Insufficient funds"

def get\_balance(self):

return f"Balance for {self.owner}: {self.\_\_balance}"

# Creating objects

acc1 = BankAccount("Alice", 1000)

print(acc1.deposit(500))

print(acc1.withdraw(200))

print(acc1.get\_balance())

**3. Python’s Version of Public, Protected, and Private Members**

**Public Members**

Accessible from anywhere. Defined normally.

class Person:

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

self.name = name # Public attribute

p = Person("Alice")

print(p.name) # Allowed

**Protected Members (\_variable)**

Accessible within the class and subclasses. Naming convention: prefix with \_.

class Person:

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

self.name = name

self.\_age = age # Protected attribute

p = Person("Alice", 30)

print(p.\_age) # Allowed but not recommended

**Private Members (\_\_variable)**

Not directly accessible outside the class. Naming convention: prefix with \_\_.

class Person:

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

self.name = name

self.\_\_salary = salary # Private attribute

p = Person("Alice", 50000)

# print(p.\_\_salary) # Will raise AttributeError

print(p.\_Person\_\_salary) # Can access using name mangling (not recommended)

**Summary**

| **Access Modifier** | **Syntax** | **Access** |
| --- | --- | --- |
| Public | self.var | Accessible everywhere |
| Protected | self.\_var | Accessible inside class & subclass (convention-based) |
| Private | self.\_\_var | Accessible inside class only (can use name mangling) |