Day-14-Object-Oriented Programming (OOP)

1. What is Object-Oriented Programming (OOP)?

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around **objects** rather than functions and logic. Objects represent real-world entities, encapsulating **data (attributes)** and **behavior (methods)** into a single unit.

Key Concepts of OOP:

- **Objects**: Instances of a class with attributes (data) and methods (functions).
- **Classes**: Blueprints/templates that define the structure and behavior of objects.
- **Encapsulation**: Restricting direct access to data and modifying it only through defined methods.
- **Abstraction**: Hiding complex implementation details and exposing only necessary parts.
- **Inheritance**: Enabling a class (child) to derive attributes and methods from another class (parent).
- **Polymorphism**: Allowing methods or functions to take different forms based on context.

---

2. Why Use Object-Oriented Programming (OOP)?

OOP makes software development more **organized, modular, scalable, and reusable**.

Advantages of OOP over Procedural Programming:

Feature	Procedural Programming	Object-Oriented Programming (OOP)
Structure	Organized in functions and procedures	Organized into objects and classes
Code Reusability	Low (Code duplication is common)	High (Classes and objects can be reused)

Feature	Procedural Programming	Object-Oriented Programming (OOP)
Data Security	Data is globally accessible	Data is encapsulated and controlled
Flexibility	Difficult to modify without affecting all functions	Changes can be made without affecting other parts
Scalability	Harder to maintain large projects	Suitable for large and complex applications

==

Use Case Example: Procedural vs OOP Approach

Procedural Approach (Without OOP)

```
# Storing data using separate variables

name = "John"

age = 30

salary = 50000


def display_employee(name, age, salary):

   print(f"Name: {name}, Age: {age}, Salary: ${salary}")


display_employee(name, age, salary)
```

- Data is scattered, making maintenance difficult.
- Functions are separate, leading to repetitive code.

==

The __init__() Function

The examples above are classes and objects in their simplest form, and are not really useful in real life applications.

To understand the meaning of classes we have to understand the built-in __init__() function.

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Note: The __init__() function is called automatically every time the class is being used to create a new object.

===

OOP Approach (Using Classes and Objects)

```
class Employee:
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}, Salary: ${self.salary}")

emp1 = Employee("John", 30, 50000)
emp1.display()
```

- **Data and behavior are encapsulated** inside the Employee class.
- The display() method is **reusable** for all Employee objects.

The __str__() Function

The __str__() function controls what should be returned when the class object is represented as a string.

If the __str__() function is not set, the string representation of the object is returned:

Example

The string representation of an object WITHOUT the __str__() function:

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

p1 = Person("John", 36)

print(p1)
```

==

Example:

The string representation of an object WITH the __str__() function:

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

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("John", 36)

print(p1)
```

Note: The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

The self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any function in the class**:**

Example

Use the words *mysillyobject* and *abc* instead of *self*:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

---

using the self keyword properly:

```
class Person:

    def __init__(self, name, age):

        self.name = name  # Using self to reference instance variables

        self.age = age


    def myfunc(self):  # Using self to refer to the instance method

        print("Hello my name is " + self.name)

p1 = Person("John", 36)

p1.myfunc()  # Output: Hello my name is John
```

Modify Object Properties

Set the age of p1 to 40:

```
p1.age = 40
```

==

Delete Object Properties

You can delete properties on objects by using the del keyword:

Example

Delete the age property from the p1 object:

```
del p1.age
```

==

Delete Objects

You can delete objects by using the del keyword:

Example

Delete the p1 object:

```
del p1
```

==

The pass Statement

class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

Example

```
class Person:
  pass
```

3. When to Use Object-Oriented Programming?

You should use OOP when:

✅ **Developing Large and Complex Applications** – OOP helps break down large projects into manageable objects.

✅ **Code Reusability is Required** – Classes can be reused in multiple parts of the application.

✅ **Security and Data Encapsulation are Needed** – Private attributes prevent accidental data modification.

✅ **Scalability is Important** – OOP makes it easier to add new features without breaking existing code.

Real-World Applications of OOP:

Industry	Example Application
Banking	Account Management, Transactions, Loan Processing
E-Commerce	Customer Profiles, Product Inventory, Shopping Carts
Healthcare	Patient Records, Hospital Management Systems
Game Development	Characters, Weapons, Levels as Objects
Machine Learning & AI	Data Models, Neural Networks

---

4. Benefits of Using OOP

Using OOP in programming offers several advantages:

✅ **1. Modularity (Code Organization)**

- OOP divides code into self-contained objects, making it easier to manage.
- Each class handles a specific part of the system.

✅ **2. Code Reusability**

- Once a class is created, it can be reused multiple times.
- Inheritance allows child classes to reuse methods and properties of the parent class.

✅ **3. Security and Data Protection (Encapsulation)**

- Data is hidden within objects and accessed only via methods.
- Prevents accidental modifications of sensitive data.

✅ **4. Scalability and Maintainability**

- OOP makes it easy to add new features without breaking existing functionality.
- Changing one class does not affect other parts of the application.

✅ **5. Flexibility (Polymorphism)**

- Objects and methods can be extended or overridden based on requirements.
- The same function can have different implementations.

Class Variables vs. Instance Variables

1. **Instance Variables**

- **Defined inside the constructor (__init__).**
- **Unique to each instance.**

2. **Class Variables**

- **Defined inside the class but outside any method.**
- **Shared among all instances of the class.**

Example of Class Variables:

```
class Employee:
  company = "TechCorp"  # Class variable (shared)

  def __init__(self, name, salary):
    self.name = name     # Instance variable
    self.salary = salary  # Instance variable

# Creating objects
emp1 = Employee("Alice", 50000)
emp2 = Employee("Bob", 60000)

print(emp1.company)  # Output: TechCorp
print(emp2.company)  # Output: TechCorp

# Changing the class variable
Employee.company = "AI Epoch"

print(emp1.company)  # Output: AI Epoch
print(emp2.company)  # Output: AI Epoch
```

🔹 **Class variables are shared among all objects, while instance variables are unique to each object.**

---

3. Static Variables (Class Variables)

- Static variables are essentially class variables.
- Stored at the class level and shared by all objects.
- Changing them via one instance affects all instances.

Example:

```
class Student:
   school = "Global High School"  # Static variable (class variable)


   def __init__(self, name, grade):
      self.name = name    # Instance variable
      self.grade = grade  # Instance variable


# Accessing static variable
print(Student.school)  # Output: Global High School


# Creating instances
s1 = Student("John", "A")
s2 = Student("Emma", "B")


print(s1.school)  # Output: Global High School
print(s2.school)  # Output: Global High School
```

# Changing static variable

Student.school = "International Academy"

print(s1.school) # Output: International Academy

print(s2.school) # Output: International Academy

---

4. Static Methods

- **Defined using @staticmethod.**
- **Do not use self or cls as parameters.**
- **Behave like normal functions inside a class.**
- **Used when we don't need access to instance or class variables.**

Example of Static Method:

```
class MathOperations:
  @staticmethod
  def add(a, b):
    return a + b

# Calling static method without creating an instance
print(MathOperations.add(5, 3))  # Output: 8
```

🔹 **Static methods are utility methods that do not depend on instance or class variables.**

---

5. Class Methods

- Defined using @classmethod.
- Take cls as the first parameter.
- Can modify class variables.

Example of Class Method:

```
class Company:
    company_name = "TechCorp"

    @classmethod
    def set_company_name(cls, new_name):
        cls.company_name = new_name  # Modifying class variable


# Accessing class method without an instance
Company.set_company_name("AI Epoch")
print(Company.company_name)  # Output: AI Epoch
```

🔹 **Class methods modify class-level variables and affect all instances.**

---

6. Access Modifiers (Public, Protected, Private)

Python does not have strict access modifiers like Java or C++. However, it follows naming conventions:

1. Public Members (Default)

- **Can be accessed from anywhere.**

```
class Person:
  def __init__(self, name):
    self.name = name  # Public variable


p = Person("Alice")
print(p.name)  # Output: Alice
```

2. Protected Members (_single_underscore)

- Conventionally indicates that the variable/method should not be accessed directly.

```
class Car:
  def __init__(self, brand):
    self._brand = brand  # Protected variable


c = Car("BMW")
print(c._brand)  # Output: BMW (but should not be accessed directly)
```

3. Private Members (__double_underscore)

- Name mangling makes it difficult to access directly.

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


  def get_balance(self):
    return self.__balance
```

```
account = BankAccount(1000)

# print(account.__balance)  # This would raise an AttributeError

print(account.get_balance())  # Output: 1000
```

🔹 **Private members are internally stored as _ClassName__variable_name, which prevents accidental modification.**

---

7. Summary Table

Feature	**Defined With**	**Accessed By**	**Special Features**
Instance Variable	self.var_name	Only instance	Unique to each instance
Class Variable	class_name.var_name	Class & all instances	Shared among all instances
Static Variable	class_name.var_name	Class & all instances	Same as class variables
Static Method	@staticmethod	Class or instance	No self or cls
Class Method	@classmethod	Class or instance	Modifies class variables
Public Variable	self.var	Everywhere	Default access
Protected Variable	_var	Within class & subclasses	Shouldn't be accessed directly
Private Variable	__var	Within class only	Uses name mangling (_ClassName__var)

---

8. Conclusion

- **Instance variables belong to objects, while class variables are shared.**
- **Static variables and class variables are the same.**
- **Static methods are utility methods.**
- **Class methods modify class variables.**
- **Access modifiers (public, protected, private) help with encapsulation.**
