OOP

OOP is a way of writing code by grouping data and functions into reusable structures called classes. It mimics real-world objects.

OOP Core Concepts:

1. Class – A blueprint for creating objects.
2. Object – A real instance based on the class.
3. Method – Function inside a class.
4. Constructor – A special method used to initialize objects.
5. Inheritance – One class can inherit features from another.

Class

A class is like a blueprint for creating objects. It defines attributes (data) and methods (functions) that objects will have.

1. class is a keyword to define a class.
2. ClassName is the name you give to the class (PascalCase is preferred).
3. The class body can have variables (attributes) and functions (methods).
4. pass means "do nothing" (used for placeholder classes).

In [7]:

```python
# Basic Syntax
class ClassName:
    # body of the class
    pass
```

Object

An object is a specific instance of a class.

In [6]:

```python
class Student:
    pass

# Creating an object
s1 = Student()
print(type(s1))  # <class '__main__.Student'>
```

```
<class '__main__.Student'>
```

1. <class '...'> → This tells you what class the object belongs to. 2. __main__ → This is the name of the current Python script you're running. So Student is defined in the main script. 3. Student → The class name of the object.

In [1]:

```python
dir()
```

```
Out[1]: ['In',
          'Out',
          '_',
          '__',
          '___',
          '__builtin__',
          '__builtins__',
          '__doc__',
          '__loader__',
          '__name__',
          '__package__',
          '__spec__',
          '_dh',
          '_i',
          '_i1',
          '_ih',
          '_ii',
          '_iii',
          '_oh',
          'exit',
          'get_ipython',
          'open',
          'quit']
```

```
In [3]: # file: example.py
        print(__name__)
```

```
__main__
```

```
In [4]: def greet():
            """This function says hello."""
            print("Hello!")

        print(greet.__doc__)
```

```
This function says hello.
```

```
In [6]: x = 42
        print(x.__class__)
```

```
<class 'int'>
```

```
In [3]: print(dir("Hello"))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq_
_', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__
getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__l
e__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '_
_str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalp
ha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isp
rintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'mak
etrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'str
ip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

In [5]:

```
help(list.append)
```

```
Help on method_descriptor:

append(self, object, /)
    Append object to the end of the list.
```

Methods

Methods define behaviors/actions the object can perform. Always use self as the first parameter.

In [11]:

```
# Methods (Functions Inside Class)
class MyClass:
    def __init__(self):
        self.name = "Taimur"

    def greet(self):
        return "Hello"

print(dir(MyClass))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format_
_', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduc
e__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subcla
sshook__', '__weakref__', 'greet']
```

In [2]:

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

    def start_engine(self):
        print(f"{self.brand} engine started!")

my_car = Car("Honda", "Blue")
my_car.start_engine()  # Output: Honda engine started!
```

```
Honda engine started!
```

Default Constructor

In [5]:

```
## Default Constructor (No Parameters)
class Bike:
    def __init__(self):
        self.brand = "Yamaha"
        self.color = "Black"

    def show_info(self):
        print(f"Brand: {self.brand}, Color: {self.color}")

bike1 = Bike()
```

```python
bike1.show_info()

# A default constructor doesn’t take any arguments (except self).
# Fixed values are assigned when the object is created.
```

```
Brand: Yamaha, Color: Black
```

In [13]:

```python
## Class and Object
# Class definition
class Car:
    def __init__(self, brand, color): #self refers to the current object.
        self.brand = brand
        self.color = color

# Creating an object
my_car = Car("Toyota", "Red")

# Accessing attributes
print(my_car.brand)  # Output: Toyota
print(my_car.color)  # Output: Red
```

```
Toyota
Red
```

In [6]:

```python
## Parameterized Constructor
class Laptop:
    def __init__(self, brand, price):
        self.brand = brand
        self.price = price

    def show_specs(self):
        print(f"Brand: {self.brand}, Price: ${self.price}")

laptop1 = Laptop("Dell", 800)
laptop1.show_specs()

# A parameterized constructor allows passing values while creating objects.
# Useful for setting dynamic values.
```

```
Brand: Dell, Price: $800
```

In [14]:

```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def show_info(self):
        print(f"Title: {self.title}, Author: {self.author}")

book1 = Book("1984", "George Orwell")
book1.show_info()
```

```
Title: 1984, Author: George Orwell
```

In [7]:

```python
## Pass Statement
class Phone:
    pass  # Placeholder for now
```

```
# pass is used when we don’t want to write any code inside a block.
# Avoids error from an empty class, method, or loop.
```

Inheritance

In [18]:

```
class Animal:
    def __init__(self):
        print("Animal is created")

    def sound(self):
        print("Animal makes a sound")

    def move(self):
        print("Animal moves")

# Child class just inherits everything from Animal
class Dog(Animal):
    pass

# Create object of Dog
d = Dog()
d.sound()
d.move()
```

```
Animal is created
Animal makes a sound
Animal moves
```

In [9]:

```
## Intro to Inheritance
# Base class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} makes a sound.")

# Derived class
class Dog(Animal):
    def speak(self):
        print(f"{self.name} barks.")

dog1 = Dog("Buddy")
dog1.speak()  # Output: Buddy barks.

# Dog inherits from Animal.
# It overrides the speak method.
# Promotes code reuse and extension.
```

```
Buddy barks.
```

Variables assigned using self.variable = value inside the parent class constructor (**init**) are not automatically inherited — but they are available if the constructor is called using super().

In [22]:

```
class A:
    def __init__(self):
        self.name = "Taimur"

class B(A):
    def __init__(self):
        super().__init__()  # Needed to inherit variables

b = B()
print(b.name)  # ✅ name is accessible after calling super().__init__()
```

```
Taimur
```

In [28]:

```
# super() lets us call the parent class constructor or method.
class Person:
    def __init__(self, name):
        self.name = name

    def show(self):
        print(f"Name: {self.name}")

class Employee(Person):
    def __init__(self, name, salary):
        super().__init__(name)  # call base constructor
        self.salary = salary

    def show(self):
        super().show()
        print(f"Salary: {self.salary}")

e1 = Employee("Alice", 50000)
e1.show()
```

```
Name: Alice
Salary: 50000
```

In [23]:

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

    def details(self):
        print(f"Name: {self.name}, ID: {self.id}")

class Manager(Employee):
    def __init__(self, name, id, department):
        super().__init__(name, id)
        self.department = department

    def details(self):
        super().details()
        #print({self.name})
        print(f"Department: {self.department}")

m = Manager("Taimur", 101, "Data Science")
m.details()
```

```
Name: Taimur, ID: 101
Department: Data Science
```

if the child class (like Employee) defines its own **init**, then it must explicitly call the parent constructor using super().**init**(...) to inherit and initialize attributes like self.name.

In [24]:

```
## Assignments
class Animal:
    def __init__(self):
        print("Animal created")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

d = Dog()
d.bark()
```

```
Animal created
Dog barks
```

In [25]:

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

class Dog(Animal):
    def show(self):
        print(f"This is {self.name}")

d = Dog("Tommy")
d.show()
```

```
This is Tommy
```

In [ ]: