Why is OOP Important?

1. OOP groups data and functions together inside classes.
2. Code is cleaner and easier to understand.
3. Once you build a class, you don't need to rewrite it.
4. New classes can inherit from old ones and add new features.
5. Increases security and prevents accidental changes.
6. Code feels more natural and logical.

OOP makes your code organized, reusable, safe, and close to real life.

Self

In [26]:
```python
class Student:
    def __init__(self, name, age):
        self.name = name    # object's name
        self.age = age      # object's age

    def greet(self):
        print(f"Hello, I am {self.name} and I am {self.age} years old.")

# Two students
s1 = Student("Alice", 20)
s2 = Student("Bob", 22)

s1.greet()
s2.greet()
```

```
Hello, I am Alice and I am 20 years old.
Hello, I am Bob and I am 22 years old.
```

These are the instance variables — they belong to the object being created.

1. self.name stores the name inside the object
2. self.age stores the age inside the object

In [31]:
```python
# Python does not require the first parameter of a method to be called self. It's j
# self is the standard naming convention in Python.
class Car:
    def __init__(this, brand):
        this.brand = brand

    def show_brand(this):
        print("Brand is:", this.brand)

c = Car("Toyota")
c.show_brand()
```

```
Brand is: Toyota
```

1. Single Inheritance

One child class inherits from one parent class.

In [ ]:
```
class Animal:
    def sound(self):
        print("Animal makes a sound")

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

# Example
d = Dog()
d.sound()  # Inherited
d.bark()   # Own method
```

2. Hierarchical Inheritance

One parent class, multiple child classes.

In [1]:
```
class Animal:
    def sound(self):
        print("Animal makes a sound")

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

class Cat(Animal):
    def meow(self):
        print("Cat meows")

# Example
d = Dog()
d.sound()
d.bark()

c = Cat()
c.sound()
c.meow()
```

```
Animal makes a sound
Dog barks
Animal makes a sound
Cat meows
```

3. Multilevel Inheritance

A child inherits from a parent, and another child inherits from that child.

In [2]:
```
class Animal:
    def sound(self):
        print("Animal sound")
```

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

class Puppy(Dog):
    def weep(self):
        print("Puppy weeps")

# Example
p = Puppy()
p.sound()
p.bark()
p.weep()
```

```
Animal sound
Dog barks
Puppy weeps
```

1. Puppy → Dog → Animal (grandchild concept)

4. Multiple Inheritance

Multiple Inheritance means a class can inherit from more than one parent class.

class ChildClass(Parent1, Parent2): # class body

In [4]:
```python
class Father:
    def skills(self):
        print("Father: Cooking, Driving")

class Mother:
    def skills(self):
        print("Mother: Painting, Gardening")

class Child(Father, Mother):
    pass

c = Child()
c.skills()
```

```
Father: Cooking, Driving
```

1. Child class inherits from both Father and Mother.
2. But since both have a method called skills(), Python uses the one from the first parent listed (Father) — this is due to the MRO (Method Resolution Order).

In [6]:
```python
# Example with Different Methods
class Engine:
    def start(self):
        print("Engine started")

class Battery:
    def charge(self):
        print("Battery charging")
```

```python
class ElectricCar(Engine, Battery):
    pass

e = ElectricCar()
e.start()   # Inherited from Engine
e.charge()  # Inherited from Battery
```

```
Engine started
Battery charging
```

In [ ]:

```python

```

In [ ]:

```python
class Father:
    def skills(self):
        print("Father: Programming")

class Mother:
    def skills(self):
        print("Mother: Painting")

class Child(Father, Mother):
    def skills(self):
        Father.skills(self)
        Mother.skills(self)
        print("Child: Dancing")

# Example
c = Child()
c.skills()
```

Methods without self

In [34]:

```python
# If you're accessing or modifying object-specific data (attributes), then self is
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):  # needs self!
        print("Hello,", self.name)
```

In [22]:

```python
# Python automatically passes the object as the first argument, so if there's no se
class WrongExample:
    #@staticmethod
    def say_hello():  # No self
        print("Hello")

obj = WrongExample()
obj.say_hello()
# TypeError: say_hello() takes 0 positional arguments but 1 was given
```

```
Hello
```

In [58]:

```python
# Typically used when the method works with or modifies the object’s attributes.
```

In [20]:
```python
#  Static Method → No self required
class MathTools:
    @staticmethod
    def add(a, b):
        return a + b

sum_ = MathTools()
sum_.add(2,3)
#print(MathTools.add(2, 3))  #  Output: 5
```

Out[20]: 5

Use @staticmethod when:

1. You don't need self or cls
2. You just want to group utility functions inside a class

In [54]:
```python
# Class Method → Uses cls instead of self

class MyClass:
    count = 0

    @classmethod
    def show_count(cls):
        print("Count is:", cls.count)
```

Use @classmethod when:

1. You want to work with class variables
2. You don't need access to individual object (self)

Decorator

They are used to wrap or decorate a function with additional functionality. For example, a decorator could add logging, timing, or access control to a function.

In [62]:
```python
# A simple decorator that prints a message before a function is called
def my_decorator(func):
    def wrapper():
        print("Before function execution")
        func()
        print("After function execution")
    return wrapper

@my_decorator  # Applying the decorator to the function
def greet():
    print("Hello!")

greet()
```

```
Before function execution
Hello!
After function execution
```

Built-in Decorators in Python

Python has several built-in decorators that serve common purposes, like:

1. @staticmethod: Used for defining static methods (doesn't access self or cls).
2. @classmethod: Used for defining class methods (accesses the class with cls).
3. @property: Used to make a method behave like an attribute (no need to call it with (), just use it like a normal attribute).

In [89]:

```python
class MathTools:
    @staticmethod
    def add(a, b):  # Static method
        return a + b

result = MathTools.add(3, 4)  # Calls the static method without needing an instance
print(result)  # Output: 7
```

7

In [83]:

```python
# class MathTools:
#     #@staticmethod
#     def add(self, a, b):  # Static method
#         self.a = a
#         self.b = b
#         return self.a + self.b

# result = MathTools()  # Calls the static method without needing an instance
# print(result.add(3,4))  # Output: 7
```

7

@property

you can access it like an attribute instead of calling it like a method. It allows you to define behavior for getting (and optionally setting) a value, without having to explicitly use method calls.

In [91]:

```python
class Circle:
    def __init__(self, radius):
        self._radius = radius  # Private attribute

    @property
    def radius(self):  # Getter method
        return self._radius

    @property
    def area(self):  # Another property method
        return 3.14159 * (self._radius ** 2)

    @radius.setter
```

```python
    def radius(self, value):  # Setter method
        if value < 0:
            raise ValueError("Radius cannot be negative.")
        self._radius = value


# Create a Circle object
circle = Circle(5)

# Access radius like an attribute
print(f"Radius: {circle.radius}")  # Calls the 'radius' property

# Access area like an attribute
print(f"Area: {circle.area}")  # Calls the 'area' property

# Set radius using the setter
circle.radius = 10
print(f"Updated Radius: {circle.radius}")
print(f"Updated Area: {circle.area}")

# Trying to set a negative radius raises an error
# circle.radius = -5  # This will raise ValueError
```

```
Radius: 5
Area: 78.53975
Updated Radius: 10
Updated Area: 314.159
```

In [103…

```python
class Circle:
    def __init__(self, radius):
        self._radius = radius  # Private attribute

    @property
    def radius(self):  # Getter method
        return self._radius

    #@property
    def area(self):  # Another property method
        return 3.14159 * (self._radius ** 2)

    @radius.setter
    def radius(self, value):  # Setter method
        if value < 0:
            raise ValueError("Radius cannot be negative.")
        self._radius = value


# Create a Circle object
circle = Circle(5)

# Access radius like an attribute
print(f"Radius: {circle.radius}")  # Calls the 'radius' property

# Access area like an attribute
print(f"Area: {circle.area()}")  # Calls the 'area' property
```

```
# Set radius using the setter
circle.radius = 10
print(f"Updated Radius: {circle.radius}")
print(f"Updated Area: {circle.area()}")

# Trying to set a negative radius raises an error
# circle.radius = -5  # This will raise ValueError
```

```
Radius: 5
Area: 78.53975
Updated Radius: 10
Updated Area: 314.159
```

Method Overloading

In [109…

```
class Math:
    def multiply(self, *args):
        result = 1
        if not args:
            return 0
        for num in args:
            result *= num
        return result

m = Math()

print(m.multiply(5))            # Output: 5
print(m.multiply(2, 3))         # Output: 6
print(m.multiply(2, 3, 4))      # Output: 24
print(m.multiply())             # Output: 0
```

```
5
6
24
0
```

In [111…

```
class Greeter:
    def greet(self, name=None):
        if name:
            print(f"Hello, {name}!")
        else:
            print("Hello, there!")

g = Greeter()
g.greet("Taimur")   # Output: Hello, Taimur!
g.greet()           # Output: Hello, there!
```

```
Hello, Taimur!
Hello, there!
```

Method Overriding

Method overriding happens when a child class provides its own version of a method that is already defined in the parent class.

In [117…

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):
    def speak(self):  # This overrides the parent method
        print("Dog barks")

# Create instances
a = Animal()
d = Dog()

# Call speak()
a.speak()   # Output: Animal makes a sound
d.speak()   # Output: Dog barks
```

```
Animal makes a sound
Dog barks
```

1. Animal has a method speak().
2. Dog is a subclass that overrides speak() with its own version.
3. When you call speak() on a Dog object, Python uses the child version, not the parent's.

In [120…

```

```

In [122…

```
class Shape:
    def area(self):
        print("Calculating area in generic shape")

class Circle(Shape):
    def area(self):
        print("Area = π × r²")

class Rectangle(Shape):
    def area(self):
        print("Area = length × width")

# Create objects
s = Shape()
c = Circle()
r = Rectangle()

s.area()  # Output: Calculating area in generic shape
c.area()  # Output: Area = π × r²
r.area()  # Output: Area = length × width
```

```
Calculating area in generic shape
Area = π × r²
Area = length × width
```

Encapsulation

1. Encapsulation is the OOP principle of hiding internal object details and restricting direct access to some parts of an object.
2. It helps protect data and makes the class more secure and manageable.

Using access modifiers:

1. public → accessible from anywhere (default in Python)
2. _protected → hint to treat as internal (not enforced)
3. __private → name mangled to restrict access from outside

Hiding private details inside a class.

In [ ]:

```
class BankAccount:
    def __init__(self):
        self.__balance = 0  # Private variable

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

    def get_balance(self):
        return self.__balance

# Example
acc = BankAccount()
acc.deposit(1000)
print(acc.get_balance())
# print(acc.__balance)  # Error: cannot access private variable
```

In [12]:

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

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited ${amount}")
        else:
            print("Invalid amount!")

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew ${amount}")
        else:
            print("Insufficient balance!")

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

In [13]:
```python
acc = BankAccount("Taimur", 1000)
acc.deposit(500)
acc.withdraw(200)
print("Balance:", acc.get_balance())
```

```
Deposited $500
Withdrew $200
Balance: 1300
```

In [15]:
```python
# Trying to access the private variable:
print(acc.__balance)  # Error!
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[15], line 2
      1 # Trying to access the private variable:
----> 2 print(acc.__balance)

AttributeError: 'BankAccount' object has no attribute '__balance'
```

In [16]:
```python
#Example of a Protected Attribute
class Person:
    def __init__(self, name, age):
        self.name = name            # Public
        self._age = age             # Protected

    def show_info(self):
        print(f"Name: {self.name}, Age: {self._age}")
```

In [17]:
```python
p = Person("Taimur", 30)
p.show_info()

# Accessing the protected attribute (still possible)
print("Accessing protected age:", p._age)
```

```
Name: Taimur, Age: 30
Accessing protected age: 30
```

1. _age is a protected attribute.
2. We can access it from outside the class, but it's considered bad practice unless you're in a subclass or need to for a good reason.

In [18]:
```python
class Student(Person):
    def is_adult(self):
        return self._age >= 18

s = Student("Ayan", 16)
print("Is adult?", s.is_adult())
# This is a more appropriate use of a protected attribute — accessed from within a
```

```
Is adult? False
```

Polymorphism

1. Polymorphism means "many forms" — in OOP, it refers to the ability to use the same method name in different classes but with different behavior.
2. It allows you to call the same method on different objects and get different results depending on the object type.

In [7]:
```
class Cat:
    def speak(self):
        print("Meow")

class Dog:
    def speak(self):
        print("Bark")

# Polymorphism in action
def animal_sound(animal):
    animal.speak()

c = Cat()
d = Dog()

animal_sound(c)  # Output: Meow
animal_sound(d)  # Output: Bark
```

```
Meow
Bark
```

Special Example of Polymorphism

In [8]:
```
class Payment:
    def pay(self, amount):
        pass

class CashPayment(Payment):
    def pay(self, amount):
        print(f"Paid ${amount} in cash.")

class CreditCardPayment(Payment):
    def pay(self, amount):
        print(f"Paid ${amount} using credit card.")

class MobilePayment(Payment):
    def pay(self, amount):
        print(f"Paid ${amount} via mobile wallet.")
```

In [9]:
```
def process_payment(payment_method, amount):
    payment_method.pay(amount)

# Create different payment method objects
cash = CashPayment()
card = CreditCardPayment()
mobile = MobilePayment()
```

In [10]:
```python
# Process payments
process_payment(cash, 100)
process_payment(card, 200)
process_payment(mobile, 150)
```

```
Paid $100 in cash.
Paid $200 using credit card.
Paid $150 via mobile wallet.
```

1. Even though all objects use the same method name (pay), they behave differently — this is polymorphism.
2. You can easily add more payment types (e.g., CryptoPayment) without changing the function.