Mastering Object-Oriented Programming with Python

Lecture 1: Why OOPs for AI/ML Engineer ?

Table of Content

1. Procedural Programming Vs Object-oriented Programming
2. Introduction to OOPs in Python
3. Writing our First Class In Python
4. Hands-On OOPs Concept : Pizza Analogy
5. Pillars of OOPs - Overview
6. Inheritance in Python
7. Encapsulation in Python
8. Polymorphism in Python
9. Data Abastraction in Python
10. Static Concept and Copy Constructor

Lecture 2: Procedural Vs OOPs

1. Programming style so far

In [1]:
```
# 1. So far we have seen Procedural Programming style of coding in python
# 2. Procedural programming organizes code using functions and procedures, followin
# 3. Even though Python doesn't have a separate procedure keyword, any function tha
# 4. Code is organized into classes and objects
# 5. Reusability - Limited (Procedural) High(OOP)
# 6. Security - Procedural ->Low (no data hiding);      OOP ->High (encapsulation s
# 7. Ideal for  Small/simple programs (Data cleaning, Sending metrics-mail)     Lar
```

In [2]:
```
"""
Problem: Student Management System

1. Add student details
2. Display student details
"""
```

Out[2]:
```
'\nProblem: Student Management System\n\n1. Add student details\n2. Display studen
t details\n'
```

In [3]:
```
# Procedural version using global list
students = []
```

```
def add_student(name, age, grade):
    students.append({'name': name, 'age': age, 'grade': grade})

def display_students():
    for s in students:
        print(f"Name: {s['name']}, Age: {s['age']}, Grade: {s['grade']}")

# Usage
add_student("Alice", 14, "8th")
add_student("Bob", 15, "9th")
display_students()
```

```
Name: Alice, Age: 14, Grade: 8th
Name: Bob, Age: 15, Grade: 9th
```

❌ **Limitations in Procedural:**

- Global variable `students` — risky in large codebases
- Data (`dict`) and behavior (`add` , `display`) are separate
- No protection — any code can change student data incorrectly
- Can't easily create multiple independent student groups

2. Object-Oriented Approach

In [5]:
```
# - Python supports both Object-oriented programming and Procedural programming app
```

In [6]:
```
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}")

class StudentManager:
    def __init__(self):
        self.students = []

    def add_student(self, name, age, grade):
        student = Student(name, age, grade)
        self.students.append(student)

    def display_all(self):
        for student in self.students:
            student.display()

# Usage
manager = StudentManager()
manager.add_student("Alice", 14, "8th")
manager.add_student("Bob", 15, "9th")
manager.display_all()
```

```
Name: Alice, Age: 14, Grade: 8th
Name: Bob, Age: 15, Grade: 9th
```

✅ Benefits of OOP in This Example

Benefit	Explanation
🔐 **Encapsulation**	Student data (`name` , `age` , `grade`) is tied with behavior (`display()`) in one object.
📦 **Modularity**	`Student` and `StudentManager` are separate, reusable components.
🧪 **Testability**	You can test `Student` and `StudentManager` independently.
🔁 **Reusability & Extension**	Can subclass `Student` (e.g., `HighSchoolStudent`) to extend behavior.
👥 **Multiple Instances**	Can create multiple independent `StudentManager` objects for different classes/schools.
💡 **Clear Real-world Mapping**	`Student` models a real student naturally, making the code easier to understand.

In [ ]:
```
# Let's look at some real world examples

# Flipkart - Class(Customer, Merchant, Product, DeliveryBoy, Employee)
# Swiggy - class(Customer, Restaurant, FoodItem, DeiveryBoy, Employee)
# Ola - class(Rider, Captain, Employee)
# Domino's - class(Customer, Outlet, PizzaItem, Employee)
```

Final Thought

Procedural programming is fine for small scripts

but OOP provides structure, safety, and scalability for real-world systems.

Lecture 3,4: Introduction to OOPs in Python

Object-Oriented Programming (OOP) is a way of organizing code by creating **objects** that represent **real-world things**.

Object-Oriented Programming (OOP) is a programming paradigm that organizes code using **objects** and **classes**.

> A **programming paradigm** is a style or approach to solving problems using code.

OOP focuses on:

- **Data (attributes)**
- **Behavior (methods)**

This promotes:

- ✅ Code reusability
- ✅ Modularity
- ✅ Easier maintenance

OOP models real-world entities as **software objects** that:

- Have some **data**
- Can perform **operations**

🧠 Think in Terms of:

- **Objects** → Real-world things
- **Classes** → Blueprints for those things
- **Properties** → Data or attributes
- **Methods** → Actions or behaviors

How Do You Define a Class in Python?

Primitive data structures—like **numbers**, **strings**, and **lists**—are designed to represent **straightforward pieces of information**, such as:

- the **cost of an apple** 🍎 (number),
- the **name of a poem** 📜 (string),
- or your **favorite colors** 🎨 (list).

But what if you want to represent something more **complex and real-world**, like:

- an **Employee** 👨‍💼,
- a **Pizza** 🍕,
- or a **Vehicle** 🚗?

❌ Using Lists or Dictionaries

We can represent such complex items using **lists** or **dictionaries**, like this:

- We can represent using different data-types like list, dict etc.

```
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

But it has some challenges.

1. First, it can make larger code files more difficult to manage.
2. It can introduce errors if employees don't have the same number of elements in their respective lists.

The best implementation is done using Classes and Objects because it makes the real world Item representation code more manageable and more maintainable

In [ ]:
```
# Problem: You need to manage a record of students taking part in different clubs -
```

In [3]:
```
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}")

class StudentManager:
    def __init__(self):
        self.students = []

    def add_student(self, name, age, grade):
        student = Student(name, age, grade)
        self.students.append(student)

    def display_all(self):
        for student in self.students:
            student.display()
```

In [10]:
```
# Usage

#Music -> (Ram, Monu)
MusicClubStudentManager = manager = StudentManager()
MusicClubStudentManager.add_student("Ram", 15, "10th")
MusicClubStudentManager.add_student("Monu", 14, "9th")
print("Members of music club: ")
MusicClubStudentManager.display_all()

#Dance -> (Sonu, Shivani)
DanceClubStudentManager = manager = StudentManager()
DanceClubStudentManager.add_student("Sonu", 15, "10th")
DanceClubStudentManager.add_student("Shivani", 16, "11th")
print("\n\nMembers of Dance club: ")
DanceClubStudentManager.display_all()

#Coding -> (Ram, Monu, Sonu, Shiavni)
CodingClubStudentManager = manager = StudentManager()
CodingClubStudentManager.add_student("Ram", 15, "10th")
CodingClubStudentManager.add_student("Monu", 14, "9th")
CodingClubStudentManager.add_student("Sonu", 15, "10th")
CodingClubStudentManager.add_student("Shivani", 16, "11th")
print("\n\nMembers of coding club: ")
CodingClubStudentManager.display_all()
```

```
#Sports -> Ram
SportsClubStudentManager = manager = StudentManager()
SportsClubStudentManager.add_student("Ram", 15, "10th")
print("\n\nMembers of sports club: ")
SportsClubStudentManager.display_all()
```

```
Members of music club:
Name: Ram, Age: 15, Grade: 10th
Name: Monu, Age: 14, Grade: 9th


Members of Dance club:
Name: Sonu, Age: 15, Grade: 10th
Name: Shivani, Age: 16, Grade: 11th


Members of coding club:
Name: Ram, Age: 15, Grade: 10th
Name: Monu, Age: 14, Grade: 9th
Name: Sonu, Age: 15, Grade: 10th
Name: Shivani, Age: 16, Grade: 11th


Members of sports club:
Name: Ram, Age: 15, Grade: 10th
```

Lecture 5,6: Writing our First Class In Python

Table of content

First class
Add member attributes
Add class attributes
create instance of the class
accessing class and instance attributes
changing attributes of class and instance and reaccessing

In [ ]:
```
# Let's Learn Class and Object in terms of Modeling a Student
# Floor Plan(BluePrint) => Flat/House(Real world Entity)
# Pseudo code => Code
```

💡 **Class Definition:**

In [1]:
```
# You start all class definitions with the class keyword, then add the name of the
# Python will consider any code that you indent below the class definition as part
# Python class names are written in CapitalizedWords notation by convention.
class Student:
    pass
```

In [5]:
```
print(type(Student))
```

```
<class 'type'>
```

MyClass is a class — but also an object.

That object is created from type — so its type is type.

💡 Adding Instance attributes:

In [15]:
```
# Add name, age, grade for Student
```

In [18]:
```
# You can give .__init__() any number of parameters, but the first parameter will a
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade
```

💡 `__init__()` method

Make sure that you indent the `.__init__()` method's signature by four spaces, and the body of the method by eight spaces. This indentation is vitally important. It tells Python that the `.__init__()` method belongs to the Student class.

Attributes created in `.__init__()` are called instance attributes. An instance attribute's value is specific to a particular instance of the class. All Student objects have a name and an age, but the values for the name and age attributes will vary depending on the Student instance.

💡 Self Parameter:

The self parameter in Python is a convention that represents the instance of the class.

It is the first parameter in instance methods and is automatically passed when calling the method.

💡 Adding Class attributes:

In [19]:
```
# add some class attributes for the Student class
```

In [20]:
```
class Student:
    # Class member variable (shared by all instances)
    school_name = "KV"

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

class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of `.__init__()`.

Feature	Class Attribute	Instance Attribute
Defined at	Class level	Inside `__init__()` or instance methods
Shared by	All instances of the class	Unique to each instance
Accessed using	`ClassName.attribute` or `object.attribute`	`object.attribute`
Best used for	Common values for all objects	Values that vary for each object

---

💡 How to instantiate a Class in Python?

Creating a new object from a class is called **instantiating a class**.

You can create a new object by typing the name of the class, followed by opening and closing parentheses:

In [23]:
```
class Student:
  pass
```

In [24]:
```
ramu = Student()
```

In [25]:
```
print(type(ramu))
```

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

In [26]:
```
print(isinstance(ramu, Student))  # True
```

```
True
```

In [14]:
```
# Every instance is an object.
# "Instance" is just a more specific word for "an object of a class."
```

In [27]:
```
Student()
```

Out[27]:
```
<__main__.Student at 0x16c9f32d6d0>
```

This funny-looking string of letters and numbers is a memory address that indicates where Python stores the Student object in your computer's memory.

In [28]:
```
Student()
```

Out[28]:
```
<__main__.Student at 0x16ca02f8e10>
```

Address is different

In [29]:
```
# Let's see something interesting
a = Student()
b = Student()
a == b
```

Out[29]:
```
False
```

Even though a and b are both instances of the Student class, they represent two distinct objects in memory.

💡 Class and Instance Attributes

In [41]:
```
class Student:
    # Class member variable (shared by all instances)
    school_name = "KV"

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

In [32]:
```
# instantiate this Computer class
Student()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[32], line 2
      1 # instantiate this Computer class
----> 2 Student()

TypeError: Student.__init__() missing 3 required positional arguments: 'name', 'ag
e', and 'grade'
```

In [44]:
```
ramu = Student("Ramu", 15, "10th") # Python creates a new instance of Student and p
```

In [35]:
```
ramu
```

Out[35]:
```
<__main__.Student at 0x16ca02f9a90>
```

💡 Access their instance attributes using dot notation

In [36]:
```
print(ramu.name)
print(ramu.age)
print(ramu.grade)
```

```
Ramu
15
10th
```

In [38]:
```
# accessing class attributes
print(ramu.school_name) # using instance of class
print(Student.school_name) # using class name
```

```
KV
KV
```

💡 Changing attribute values of an instance and class

In [45]:
```python
#changing the Instance attributes
print(f'old name is {ramu.grade}')
ramu.grade = "11th"
print(f'new name is {ramu.grade}')
```

```
old name is 10th
new name is 11th
```

In [46]:
```python
# change the class attributes
print(f'old name is {ramu.school_name}')
ramu.school_name = "KV NEW"
print(f'new name is {ramu.school_name}')
```

```
old name is KV
new name is KV NEW
```

In [47]:
```python
# No change in Class attribute value
print(Student.school_name)

# Class attribute value change in above case is very specific to object
print(ramu.school_name)
```

```
KV
KV NEW
```

The key takeaway here is that custom objects are mutable by default.

💡 Instance Methods

Instance methods are functions that you define inside a class and can only call on an instance of that class. Just like . `__init__()` , an instance method always takes self as its first parameter.

In [49]:
```python
class Student:
    # Class member variable (shared by all instances)
    school_name = "KV"

    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}")
```

In [50]:
```python
# Creating two objects (instances)
rahul = Student("Rahul", 15, "10th")
aman = Student("Aman", 16, "11th")
```

In [52]:
```
# Displaying their info
rahul.display()
aman.display()
```

```
Name: Rahul, Age: 15, Grade: 10th
Name: Aman, Age: 16, Grade: 11th
```

In [53]:
```
print(rahul)
```

```
<__main__.Student object at 0x0000016C9F502510>
```

When you print rahul, you get a cryptic-looking message telling you that rahul is a Student object at the memory address 0x0000016C9F502510.

You can change what gets printed by defining a special instance method called `.__str__()`.

In [54]:
```
class Student:
    # Class member variable (shared by all instances)
    school_name = "KV"

    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def __str__(self):
        return f"Your Details: name: {self.name}, age: {self.age}, Grade: {self.gra

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}")
```

In [56]:
```
rahul = Student("Rahul", 15, "10th")
print(rahul)
```

```
Your Details: name: Rahul, age: 15, Grade: 10th
```

Methods like `.__init__()` and `.__str__()` are called dunder methods because they begin and end with double underscores. There are many dunder methods that you can use to customize classes in Python.

💡 Class Methods

In [3]:
```
class Student:
    # Class member variable (shared by all instances)
    school_name = "KV"

    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade

    def __str__(self):
        return f"Your Details: name: {self.name}, age: {self.age}, Grade: {self.gra
```

```
    def display(self):
        print(f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}")

    @classmethod
    def change_school(cls, new_name):
        cls.school_name = new_name
```

In [4]:
```
aman =  Student("Aman",15,"11th")
```

In [5]:
```
aman.change_school("KV NEW 2")
```

In [6]:
```
print(aman)
```

```
Your Details: name: Aman, age: 15, Grade: 11th
```

In [67]:
```
Student.change_school("KV New")
```

In [68]:
```
aman.school_name
```

Out[68]:
```
'KV New'
```

✅ cls lets the method access and modify class-level data.

💡 Object as Parameter

In Python, you can pass objects as parameters to functions or methods, allowing you to manipulate or interact with those objects within the function.

When an object is passed as a parameter, the function receives a reference to the object, allowing it to access and modify the object's attributes.

When you pass an object as a parameter, you're passing a reference to the object, so any changes made to the object's properties within the method will affect the original object outside the method as well.

This is because the reference points to the same memory location where the object's data is stored.

In [58]:
```
class Student:
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade

    def display(self):
        print(f"Name: {self.name}")
        print(f"Grade: {self.grade}")

# Function that accepts a Student object
def print_student_details(student_obj):
    student_obj.display()  # Accessing method of the passed object
```

```
# Create a Student object
s1 = Student("Sanjeev", "A-")

# Pass the object to the function
print_student_details(s1)
```

```
Name: Sanjeev
Grade: A-
```

In [60]:
```
def upgrade_grade(student_obj):
    student_obj.grade = "A+"
    print(f"{student_obj.name}'s grade upgraded!")

upgrade_grade(s1)
s1.display()
```

```
Sanjeev's grade upgraded!
Name: Sanjeev
Grade: A+
```

Lecture 7: Access Modifiers in Python

Quick Recap of last lecture

```
First class
Added Instance attributes and class attributes
Added Instance methods and class methods
create instance of the class
accessing class and instance attributes
```

Access modifiers can (and should) be used for methods in object-oriented programming, including in Python, to control who can call the methods—just like with attributes.

Python does not have explicit keywords like "public," "private," or "protected".

Instead, It relies on naming conventions to indicate the intended visibility

---

1. Public (default): Members are accessible from anywhere, both within the class and outside the class.
2. Protected (_single): Members are accessible within the class, within derived classes, and within the same module. However, they are considered conventionally private, and their use outside the class or module is discouraged.
3. Private (__double): Members are accessible only within the class. They are not accessible in derived classes or outside the class.

In [24]:
```
class Student:
    # Class member variable (shared by all instances)
```

```
    school_name = "KV"

    def __init__(self, name, age, grade):
        self.__name = name
        self.age = age
        self.grade = grade

    def __str__(self):
        return f"Your Details: name: {self.__name}, age: {self.age}, Grade: {self.g

    def display(self):
        print(f"Name: {self.__name}, Age: {self.age}, Grade: {self.grade}")

    @classmethod
    def change_school(cls, new_name):
        cls.school_name = new_name
```

In [25]:
```
# Without access modifier - or default case
ramu = Student("Ramu",15,"10th")
print(ramu)
```

```
Your Details: name: Ramu, age: 15, Grade: 10th
```

In [30]:
```
ramu.__name
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[30], line 1
----> 1 ramu.__name

AttributeError: 'Student' object has no attribute '__name'
```

NOTE: usually happens because you're trying to access a private attribute that is name-mangled by Python.

In [12]:
```
#updating ramu - name
ramu.name = "Ramu Singh"
print(ramu)
```

```
Your Details: name: Ramu Singh, age: 15, Grade: 10th
```

In [22]:
```
ramu.name
```

Out[22]:
```
'Ramu Singh'
```

In [23]:
```
# After making name attribute as private
ramu.__name = "Ramu Singh"
print(ramu)
```

```
Your Details: name: Ramu, age: 15, Grade: 10th
```

Types of Access Modifiers in Python (Convention Based)

Modifier	Syntax	Accessibility
Public	`name`	Accessible everywhere

Modifier	Syntax	Accessibility
Protected	`_name`	Accessible in class & subclass (by convention)
Private	`__name`	Not accessible outside class directly (name mangled)

Why do we need Access Modifier ?

1. Encapsulation and Data Hiding

- Access modifiers restrict access to internal object details.
- They protect internal states from being modified accidentally or maliciously.

In [31]:
```
class BankAccount:
    def __init__(self):
        self.__balance = 0  # private attribute

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

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

In [34]:
```
account = BankAccount()
account.deposit(1000)
print(account.get_balance())  # ✅ Output: 1000

print(account.__balance)      # ❌ Error: AttributeError
```

```
1000
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[34], line 5
      2 account.deposit(1000)
      3 print(account.get_balance())  # ✅ Output: 1000
----> 5 print(account.__balance)

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

NOTE: Behind the scenes, Python renames __balance to _BankAccount__balance.

2. Improved Security

- Prevents sensitive data from being exposed.
- Ensures only authorized parts of the code can access or change data.

In [35]:
```
class User:
    def __init__(self, username, password):
        self.username = username
        self.__password = password  # private
```

```
    def check_password(self, input_password):
        return self.__password == input_password
```

In [36]:
```
u = User("admin", "1234")

print(u.username)              # ✅ Accessible
print(u.check_password("1234"))  # ✅ Output: True
print(u.__password)            # ❌ Error: AttributeError
```

```
admin
True
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[36], line 5
      3 print(u.username)              # ✅ Accessible
      4 print(u.check_password("1234"))  # ✅ Output: True
----> 5 print(u.__password)

AttributeError: 'User' object has no attribute '__password'
```

3. Control Over Code Behavior

- You can define how and when an object's internal state is modified.
- Example: only allow updating age if the new age is valid.

In [37]:
```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.__age = None
        self.set_age(age)

    def set_age(self, age):
        if age >= 0:
            self.__age = age
        else:
            print("Invalid age")

    def get_age(self):
        return self.__age
```

In [38]:
```
s = Student("John", 20)
print(s.get_age())    # ✅ Output: 20

s.set_age(-5)         # ⚠️ Output: Invalid age
print(s.get_age())    # ✅ Output: 20
```

```
20
Invalid age
20
```

4. Ease of Maintenance

- Minimizes bugs by isolating code changes.

- If implementation changes, the internal logic can be updated without affecting code that uses the class externally.

In [39]:
```
class Employee:
    def __init__(self, name, basic_salary):
        self.name = name
        self.__basic_salary = basic_salary

    def get_salary(self):
        # Initially, salary = basic salary
        return self.__basic_salary

e = Employee("Alice", 50000)
print(e.get_salary())  # Output: 50000
```

```
50000
```

In [40]:
```
    def get_salary(self):
        # Now salary includes a 10% bonus
        return self.__basic_salary + (0.10 * self.__basic_salary)
```

Access Modifiers for Methods in Python

Modifier	Syntax	Meaning
Public	`def method()`	Accessible from anywhere
Protected	`def _method()`	Intended for internal or subclass use
Private	`def __method()`	Not accessible directly (name mangling)

In [41]:
```
class Secret:
    def __secret_method(self):
        print("This is private!")

    def access_secret(self):
        self.__secret_method()

s = Secret()
s.access_secret()  # ✅ Works
s.__secret_method()  # ❌ AttributeError
```

```
This is private!
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[41], line 10
      8 s = Secret()
      9 s.access_secret()  # ✅ Works
---> 10 s.__secret_method()

AttributeError: 'Secret' object has no attribute '__secret_method'
```

Internally, __secret_method becomes _Secret__secret_method.

SUMMARY :

While Python doesn't enforce strict access control like Java or C++, it uses naming conventions to simulate access modifiers for methods:

In case of Inheritance

Member Visibility	Public (default)	Protected (_single)	Private (__double)
In Base Class	Accessible	Accessible	Accessible
In Derived Class	Accessible	Accessible within subclass/module	Not Accessible

> **Python does not enforce strict access control. It relies on conventions and developer discipline.**

Lecture 8: Getter and Setters in Python

Quick Recap of last lecture

```
Python does not have explicit keywords like "public," "private," or
"protected".
Instead, It relies on naming conventions to indicate the intended
visibility.
name(public), _name(protected), __name(private)
Access modifier is applicable for both attributes and methods
Python doesn't enforce strict access control like Java or C++, it uses
naming conventions to simulate access modifiers for methods
```

In Python, getters and setters are used to access and modify private attributes of a class in a controlled way. They are part of encapsulation, a core OOP principle that helps in maintaining clean, modular, and safe code.

AI/ML usecase of Getters and Setters

- Setting Input data Shape

Let's see getters and setters in simplest way

In [20]:
```python
import numpy as np

class DataLoader:
    def __init__(self):
        self._data = None  # internal attribute

    def get_data(self):
        return self._data
```

```
    def set_data(self, arr):
        if not isinstance(arr, np.ndarray):
            raise TypeError("Data must be a NumPy array.")
        if arr.ndim != 2:
            raise ValueError("Input data must be 2-dimensional.")
        print(f"Data shape set to: {arr.shape}")
        self._data = arr
```

In [ ]:
```
# In real ML pipelines, the wrong shape can crash models or silently produce incorr
# Catching errors early (during data loading) avoids wasting time on failed or flaw
# You won't have to debug mysterious shape mismatch errors deep in model code.
```

In [21]:
```
loader = DataLoader()

# Use setter explicitly
loader.set_data(np.ones((3, 4)))

# Use getter explicitly
print(loader.get_data())
```

```
Data shape set to: (3, 4)
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

Let's see getters and setters using `property()`

In [22]:
```
# Let's see getters and setters using property()
# Name the methods as get_<attribute> and set_<attribute>
import numpy as np

class DataLoader:
    def __init__(self):
        self._data = None # protected - intended to be private (convention, not enf

    # This method is the getter for the _data attribute.
    # When someone accesses loader.data, it internally calls get_data() and returns
    def get_data(self):
        return self._data

    # This is the setter method. It allows you to set the value of data (with valid
    # It takes one argument: arr — the new data to assign.
    def set_data(self, arr):
        if not isinstance(arr, np.ndarray):
            raise TypeError("Data must be a NumPy array.")
        if arr.ndim != 2:
            raise ValueError("Input data must be 2-dimensional.")
        print(f"Data shape set to: {arr.shape}")
        self._data = arr

    # Create property
    data = property(get_data, set_data)
```

In [23]:
```
loader = DataLoader()

# Set data
loader.data = np.ones((3, 4))  # ✅ Works

# Get data
print(loader.data)             # ✅ Returns the array

# Invalid input
# loader.data = [1, 2, 3]      # ❌ Raises TypeError
# loader.data = np.ones(3)     # ❌ Raises ValueError
```

```
Data shape set to: (3, 4)
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

In [ ]:
```
# data = property(get_data, set_data)
# This is the key line that turns get_data() and set_data() into a "property".
# This allows you to use loader.data like an attribute, while still having getter a
# It's equivalent to using @property and @data.setter decorators, but defined expli
```

`property()` is a built-in Python function that lets you bind getter, setter, and deleter functions to a class attribute. It allows you to control attribute access in an object-oriented and Pythonic way.

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Argument	Meaning
`fget`	Function to **get** the attribute value
`fset`	Function to **set** the attribute value
`fdel`	Function to **delete** the attribute
`doc`	Optional docstring for the property

Let's see getters and setters using `@property()`

In [17]:
```
# Let's see Pythonic way of implementation
import numpy as np

class DataLoader:
    def __init__(self):
        self._data = None

    @property
    def data(self):
        return self._data

    @data.setter
    def data(self, arr):
        if not isinstance(arr, np.ndarray):
            raise TypeError("Data must be a NumPy array.")
```

```
        if arr.ndim != 2:
            raise ValueError("Input data must be 2-dimensional.")
        print(f"Data shape set to: {arr.shape}")
        self._data = arr
```

In [18]:
```
# Example usage
loader = DataLoader()
loader.data = np.random.rand(100, 20)  # OK
# loader.data = [1, 2, 3]              # Raises TypeError
# loader.data = np.random.rand(100)    # Raises ValueError
```

```
Data shape set to: (100, 20)
```

In [5]:
```
# In real ML pipelines, the wrong shape can crash models or silently produce incorr
# Catching errors early (during data loading) avoids wasting time on failed or flaw
# You won't have to debug mysterious shape mismatch errors deep in model code.
```

The `@` symbol is used to apply a decorator to a function or method.

A decorator is a function that:

- Takes another function or method as input
- Adds some extra behavior
- Returns a modified function or method

`property()` Vs `@property`

Feature	`property()`	`@property`
Flexibility	Explicit and good for dynamic property creation	More readable, idiomatic in Python
Use case	Legacy code, dynamic behavior	Preferred in modern OOP Python
Components	`fget`, `fset`, `fdel`, `doc`	`@property`, `@<name>.setter`, `@<name>.deleter`

Lecture 9: Decorator in Python

A decorator is just a function that takes another function and adds extra behavior to it — without changing the original function's code.

Think of it like wrapping a gift 🎁 — the gift is still inside, but now it has something extra on top.

Hello-World Decorator Program

In [7]:
```
def say_hello():
    print("Hello, world!")
```

In [8]:
```
def my_decorator(func): # my_decorator is a function that takes another function (f
    def wrapper(): # Inside, it defines a wrapper function that:
        print("Before the function runs")
        func()
        print("After the function runs")
    return wrapper # Then it returns the wrapper, not the original
```

In [9]:
```
@my_decorator
def say_hello(): # say_hello = my_decorator(say_hello)
    print("Hello, world!")
```

In [10]:
```
say_hello()
```

```
Before the function runs
Hello, world!
After the function runs
```

In [11]:
```
import numpy as np

class DataLoader:
    def __init__(self):
        self._data = None

    @property
    def data(self):
        return self._data

    @data.setter
    def data(self, arr):
        if not isinstance(arr, np.ndarray):
            raise TypeError("Data must be a NumPy array.")
        if arr.ndim != 2:
            raise ValueError("Input data must be 2-dimensional.")
        print(f"Data shape set to: {arr.shape}")
        self._data = arr

# Example usage
loader = DataLoader()
loader.data = np.random.rand(100, 20)  # OK
# loader.data = [1, 2, 3]              # Raises TypeError
# loader.data = np.random.rand(100)    # Raises ValueError
```

```
Data shape set to: (100, 20)
```

The `@` symbol is used to apply a decorator to a function or method.

A decorator is a function that:

- Takes another function or method as input
- Adds some extra behavior
- Returns a modified function or method

In Your Code: `@property` **and** `@data.setter` These decorators are used to define getters and setters for the data attribute of your DataLoader class.

Step-by-Step Breakdown

In [ ]:
```python
# 1. @property
@property
def data(self):
    return self._data

# This makes data() behave like a read-only attribute.
# So now, you can access loader.data instead of calling loader.data().
```

In [ ]:
```python
# 2. @data.setter

@data.setter
def data(self, arr):
    if not isinstance(arr, np.ndarray):
        raise TypeError("Data must be a NumPy array.")
    if arr.ndim != 2:
        raise ValueError("Input data must be 2-dimensional.")
    print(f"Data shape set to: {arr.shape}")
    self._data = arr

# This decorates the method to be the setter for the data property.
# When you write loader.data = ..., it calls this method automatically.
# Here, it performs type and shape checks, then assigns to _data.
```

`property()` **Vs** `@property`

Feature	`property()`	`@property`
Flexibility	Explicit and good for dynamic property creation	More readable, idiomatic in Python
Use case	Legacy code, dynamic behavior	Preferred in modern OOP Python
Components	`fget` , `fset` , `fdel` , `doc`	`@property` , `@<name>.setter` , `@<name>.deleter`

Lecture 10: Dunder Methods in Python

In [ ]:
```python
# private attributes - getters and setters
# 3 ways to implement - trivial, property(), @property
# decorators - functions that takes another function and adds extra behavior to it
```

In [ ]:
```
__init__        Object constructor
__str__ String representation (print(obj))
```

```
__getitem__     Indexing (obj[0])
__setitem__     Assignment to index (obj[1] = x)

__iter__()      Returns the iterator object itself
__next__() Returns the next value in the sequence
```

Dunder methods (short for double underscore methods) are special methods in Python with names that start and end with double underscores, like `__init__`, `__str__`, etc.

They're also known as:

1. Magic methods
2. Special methods

These are special methods that let you customize the behavior of your objects when they interact with built-in Python syntax, operators, or functions.

They're called "dunder" because their names start and end with double underscores, like **init**, **str**, or **add**.

Dunder methods "hook into" Python's built-in behaviors. Let's look into few examples:

1. Want to define how your object looks when printed? Use `__str__()`.

In [ ]:
```python
# without __str__()
class Book:
    def __init__(self, title):
        self.title = title

b = Book("1984")
print(b)   # Output: Book: 1984
```

In [ ]:
```python
# with __str__()
class Book:
    def __init__(self, title):
        self.title = title

    def __str__(self):
        return f"Book: {self.title}"

b = Book("1984")
print(b)   # Output: Book: 1984
```

2. Want to define what happens when someone uses + on your object? Use `__add__()`.

In [ ]:
```python
# without __add__()
class Vector:
    def __init__(self, x, y):
        self.x = x
```

```
        self.y = y

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(1, 2)
v2 = Vector(3, 4)
print(v1 + v2)  # Output: Vector(4, 6)
```

In [ ]:
```
# with __add__()
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(1, 2)
v2 = Vector(3, 4)
print(v1 + v2)  # Output: Vector(4, 6)
```

3. Want to control how len(obj) behaves? Use `__len__()`.

In [ ]:
```
# without __len__()
class Basket:
    def __init__(self, items):
        self.items = items


b = Basket(['apple', 'banana'])
print(len(b))   # Output: 2
```

In [ ]:
```
# with __len__()
class Basket:
    def __init__(self, items):
        self.items = items

    def __len__(self):
        return len(self.items)

b = Basket(['apple', 'banana'])
print(len(b))   # Output: 2
```

Some Common Dunder Methods

Dunder Method	Purpose
`__init__`	Object constructor
`__str__`	String representation (`print(obj)`)

Dunder Method	Purpose
`__repr__`	Official string for debugging
`__len__`	Length (`len(obj)`)
`__getitem__`	Indexing (`obj[0]`)
`__setitem__`	Assignment to index (`obj[1] = x`)
`__eq__` , `__lt__` , etc.	Comparisons (`==` , `<` , etc.)
`__add__` , `__sub__` , etc.	Arithmetic operations

What Does `__add__()` Do?

In [ ]:
```
# In Python, when you use the + operator like this: a + b
# Python internally tries to call: a.__add__(b)
# So, if you define a custom class and implement the __add__() method, you are tell
# of your class. This is called operator overloading. -- Runtime Polymorphism
# Python has default behavior for the + operator only for built-in types (like inte
# Your class doesn't support + by default. When you implement __add__(), you are pr
# So you're not overwriting existing behavior, but rather:
# 1. Adding support for + in your class
# 2. Overriding the default "unsupported operand" error
```

Bonus:

`__iter__()` and `__next__()` are magic methods (also called dunder methods), just like `__add__()` and `__str__()` . These two are specifically used to make your object iterable, so it can be used in a for loop or with functions like `next()` .

What they Do?

Method	Purpose
`__iter__()`	Returns the iterator object itself
`__next__()`	Returns the next value in the sequence

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Lecture 11: Static variables and Methods in Python

Static concepts in python

In Python, the concept of **"static"** is not as explicit as in other languages like **Java**. However, similar behavior can be achieved using:

1. Static Variables (Class Attributes): In Python, you can use class attributes to simulate static variables shared among all instances of a class.
2. Static methods : You can use the `@staticmethod` decorator to define static methods that don't require access to the instance.

> 💡 While Python is dynamic and flexible, you can still apply static-like patterns when needed for utility code or shared logic.

Static Variables in Python

Static variables are class-level variables — shared across all instances of the class.

We Implement Static Variable concept with - **Class Attributes**

In [58]:
```
# Static Variables
class Car:
    wheels = 4  # Static variable (class attribute)

    def __init__(self, brand):
        self.brand = brand  # Instance attribute

# Accessing
car1 = Car("Toyota")
car2 = Car("Honda")
```

In [4]:
```
# before any update
print(car1.wheels)  # 4
print(car2.wheels)  # 4
print(Car.wheels) # 4
```

```
4
4
4
```

In [6]:
```
# updating using class name - and then reading again
Car.wheels = 6

#after updates
```

```
print(car1.wheels)  # 6
print(car2.wheels)  # 6
print(Car.wheels) # 6
```

```
6
6
6
```

- If you change wheels using class name, it updates for all.

In [9]:
```
# updating using instance name - and then reading again
car1.wheels = 8

#after updates
print(car1.wheels)  # 4
print(car2.wheels)  # 6
print(Car.wheels) # 6
```

```
8
6
6
```

- But if you do car1.wheels = 8, it creates a new instance variable for car1 only.
- **When you update a class attribute using an instance, Python does not actually update the class aatribute — instead, it creates a new instance attribute that shadows the class attribute only for that instance.**

Static Methods in Python

A `@staticmethod` in Python is a method inside a class that does not operate on an instance (`self`) or class (`cls`), and behaves like a regular function but lives inside a class for logical grouping.

In [13]:
```
#Static Methods
class MyClass:
    @staticmethod
    def static_method():
        print("This is a static method.")

# Calling the static method
MyClass.static_method()
```

```
This is a static method.
```

A static method is a method that belongs to a class but does not take `self` or `cls` as the first argument.

- Declared using `@staticmethod` decorator.
- Can be called on the class or on an instance.
- Cannot access or modify class or instance attributes.
- It's just a regular function that is namespaced inside the class.

When to Use?

- Use when a method doesn't need access to instance (`self`) or class (`cls`) variables.

Static Methods Usecase

1. Utility fuctions
2. Factory Methods

In [14]:
```python
# static method - as a utility functions
# Perform helper tasks that are logically related to a class, but don't require acc

class MathHelper:
    @staticmethod
    def is_even(n):
        return n % 2 == 0

    @staticmethod
    def square(x):
        return x * x

# Usage
print(MathHelper.is_even(10))   # True
print(MathHelper.square(4))     # 16
```

```
True
16
```

In [ ]:
```python
# static method - as Factory Methods
# A factory method is a method that creates and returns new instances (objects), of
# of how those instances are created.
# 1. Control or customize object creation.
# 2. Hide or abstract construction logic.
# 3. Useful for validations, alternative constructors, or multiple input formats.

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

    @staticmethod
    def create(name, age):
        if age < 18:
            raise ValueError("User must be at least 18 years old")
        return User(name, age)

# Usage
user1 = User.create("Alice", 25)
print(user1.name)  # Alice
```

NOTE:

Python does not support true private constructors like Java or C++

So as per Python, while you can't make a constructor (`__init__`) private, you can restrict or discourage its direct use.

Static Vs Class Vs Instance Methods

In [11]:
```python
class Person:
    species = "Homo sapiens"  # Class attribute

    def __init__(self, name, age):
        self.name = name        # Instance attribute
        self.age = age

    # Instance Method: has access to self (object)
    def greet(self):
        return f"Hi, I'm {self.name} and I'm {self.age} years old."

    # Class Method: has access to cls (class)
    @classmethod
    def get_species(cls):
        return f"We are {cls.species}."

    # Static Method: no access to self or cls
    @staticmethod
    def is_adult(age):
        return age >= 18
```

In [12]:
```python
# Creating an instance
p1 = Person("Alice", 22)

# 1. Instance method call → works on an object
print(p1.greet())          # Hi, I'm Alice and I'm 22 years old.

# 2. Class method call → can be called via class or object
print(Person.get_species()) # We are Homo sapiens
print(p1.get_species())     # Also works

# 3. Static method call → no object/class state used
print(Person.is_adult(22))  # True
print(p1.is_adult(15))      # Also works
```

```
Hi, I'm Alice and I'm 22 years old.
We are Homo sapiens.
We are Homo sapiens.
True
False
```

Summary Table

Feature	Instance Method	Class Method	Static Method
Decorator	*(None)*	`@classmethod`	`@staticmethod`
First Arg	`self`	`cls`	No special first argument

Feature	Instance Method	Class Method	Static Method
Access instance vars	✅ Yes	❌ No	❌ No
Access class vars	✅ Yes	✅ Yes	❌ No
Typical Use Case	Work with instance data	Work with class-level data	Utility functions, factory methods

---

Lecture 12: Constructor & Object Creation in Python

A constructor is a special method that is automatically called when an object is created. It is used to initialize the instance variables (attributes) of the class.

Python technically has two special methods related to object creation:

Method	Purpose
`__new__()`	Creates a new instance (allocates memory)
`__init__()`	Initializes the new instance

Step-by-Step Flow of Object Creation

Let's understand how object creation works in Python:

Step 1: `__new__` is called

- It creates and returns a new instance.
- It is a static method of the class (often inherited from object).
- Rarely overridden unless you're doing advanced work (like with immutable types or metaclasses).

Step 2: `__init__` is called

- It initializes the instance returned by `__new__`.
- It sets up initial values (state) of the object.

In [53]:
```
#  Example 1: Basic Constructor with __init__
class Student:
    def __init__(self, name, roll):
        print("Inside __init__")
        self.name = name
        self.roll = roll

s1 = Student("Sanjeev", 101)
print(s1.name, s1.roll)
```

```
Inside __init__
Sanjeev 101
```

In [54]:
```
"""
In above Example 1
Only __init__ is defined.
__new__ is implicitly used (inherited from object).
"""
```

Out[54]:
```
'\nIn above Example 1\nOnly __init__ is defined.\n__new__ is implicitly used (inhe
rited from object).\n'
```

In [55]:
```
# Example 2: Using __new__ and __init__ Together
class Student:
    def __new__(cls, *args, **kwargs):
        print("Inside __new__")
        instance = super().__new__(cls)
        return instance

    def __init__(self, name, roll):
        print("Inside __init__")
        self.name = name
        self.roll = roll

s1 = Student("Sanjeev", 202)
```

```
Inside __new__
Inside __init__
```

In [ ]:
```
"""
In above examples
__new__ is called first → creates the object.
__init__ is called next → initializes the object.
"""
```

Python does not support private constructors like Java/C++, but you can simulate it using `__new__()`

It means you can't prevent someone from writing Singleton(). But we can add logic in **new**() that a new object won't be created second time. so it behaves like a private constructor.

In [56]:
```
# Problem 1: To create a class (Singleton) that ensures only one object is created,
# no matter how many times the class is instantiated

class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            print("Creating new instance")
            # super() gives you access to methods from the base (parent) class.
            # In this case, the base class is object (the root of all classes in Py
            # So, super().__new__(cls) is calling: object.__new__(cls)
            # which creates a fresh instance of the class cls.
            # cls is the class itself (not an instance).
```

```
            # _instance is a class variable used to store the one and only object.
            cls._instance = super().__new__(cls)
        return cls._instance

a = Singleton()
b = Singleton()
print(a is b)  # True
```

```
Creating new instance
True
```

Explanation

✅ __new__ controls object creation
Python calls __new__ before __init__.
It creates the object and returns it.

✅ Singleton logic
cls._instance is a class variable used to store the one-and-only instance.
First time: _instance is None, so:
    super().__new__(cls) creates a new object.
    It is saved in _instance.
Subsequent times: _instance is already set, so:
    The same object is returned again.

✅ a is b
Both a and b refer to the same memory location.
print(a is b) → True

Lecture 13: Copy Constructor in Python

Copy Constructor

A copy constructor is a special constructor that creates a new object by copying the attributes of an existing object.

1. In Python, you can implement a copy constructor using a special method called `__copy__`
2. Using `copy` module

In [45]:
```
"""
Method 1: Custom Copy Constructor
1. In Python, you can implement a copy constructor using a special method called __
2. The __copy__ method in Python is a special (magic) method used to control how a
copy.copy() function.
3. creates a shallow copy
"""

class Student:
    def __init__(self, name, marks):
```

```
        self.name = name
        self.marks = marks

    # Copy constructor
    def __copy__(self):
        new_object = type(self)(self.name, self.marks)
        return new_object
```

In [46]:
```
# Creating an object of MyClass
original_obj = Student("Sanjeev", [90,80])

# Using the copy constructor to create a new object
copied_obj = original_obj.__copy__()

# before update
print("Original Object: name={}, marks={}".format(original_obj.name, original_obj.m
print("Copied Object: name={}, marks={}".format(copied_obj.name, copied_obj.marks))

#updating the value
original_obj.name = "Raju"
original_obj.marks.append(20)

# after update
print("\nOriginal Object: name={}, marks={}".format(original_obj.name, original_obj
print("Copied Object: name={}, marks={}".format(copied_obj.name, copied_obj.marks))
```

```
Original Object: name=Sanjeev, marks=[90, 80]
Copied Object: name=Sanjeev, marks=[90, 80]

Original Object: name=Raju, marks=[90, 80, 20]
Copied Object: name=Sanjeev, marks=[90, 80, 20]
```

In [47]:
```
print(f"Original Object's name ID: {id(original_obj.name)}")
print(f"Original Object's marks ID: {id(original_obj.marks)}")

print(f"\nCopied Object's name ID: {id(copied_obj.name)}")
print(f"Copied Object's marks ID: {id(copied_obj.marks)}")
```

```
Original Object's name ID: 2377827420752
Original Object's marks ID: 2377828407936

Copied Object's name ID: 2377814378416
Copied Object's marks ID: 2377828407936
```

In [49]:
```
"""
Method 2: Using copy Module
Python provides a copy module with two functions:

copy.copy() → Shallow copy
copy.deepcopy() → Deep copy

"""

import copy

class Student:
    def __init__(self, name, marks):
```

```
        self.name = name
        self.marks = marks

#creating an instance
s1 = Student("Sanjeev", [90, 80])
# Shallow copy
s2 = copy.copy(s1)
# Deep copy
s3 = copy.deepcopy(s1)
```

In [51]:
```
print(f"s1(original Instance) details- name:{s1.name} at {id(s1.name)}, marks:{s1.m
print(f"s2(Shallow Instance) details- name:{s2.name} at {id(s2.name)}, marks:{s2.ma
print(f"s3(deep Instance) details- name:{s3.name} at {id(s3.name)}, marks:{s3.marks

# Changing data in s1 (Original Instance)
s1.marks.append(100)
s1.name = "Raju"

print(f"\ns1(original Instance) details- name:{s1.name} at {id(s1.name)}, marks:{s1
print(f"s2(Shallow Instance) details- name:{s2.name} at {id(s2.name)}, marks:{s2.ma
print(f"s3(deep Instance) details- name:{s3.name} at {id(s3.name)}, marks:{s3.marks
```

```
s1(original Instance) details- name:Raju at 2377827420752, marks:[90, 80, 100] at 23
77828341888
s2(Shallow Instance) details- name:Sanjeev at 2377814378416, marks:[90, 80, 100] at
2377828341888
s3(deep Instance) details- name:Sanjeev at 2377814378416, marks:[90, 80] at 23778284
07360

s1(original Instance) details- name:Raju at 2377827420752, marks:[90, 80, 100, 100]
at 2377828341888
s2(Shallow Instance) details- name:Sanjeev at 2377814378416, marks:[90, 80, 100, 10
0] at 2377828341888
s3(deep Instance) details- name:Sanjeev at 2377814378416, marks:[90, 80] at 23778284
07360
```

Lecture 14: Hands-On OOPs Concept : Pizza Analogy

We'll learn the key concepts of OOP with the example of a **Pizza**.

Class → Blueprint

Think of a **Pizza Recipe** as a class. It's just the **instructions**, not a real pizza.

It's a **blueprint** that tells you how to make a pizza — what ingredients to use and how to cook it.

```
class Pizza:
    def __init__(self, size, toppings):
        self.size = size
        self.toppings = toppings
```

```
    def bake(self):
        print(f"Baking a {self.size} pizza with {',
'.join(self.toppings)}.")

    def serve(self):
        print("Pizza is ready to serve!")
```

Object → Actual Pizza

Now when you follow the recipe and actually make a pizza, that's an object.

```
my_pizza = Pizza("Medium", ["Cheese", "Olives"])
```

Attributes → Pizza Details

These are like the **ingredients or properties** of the pizza:

- Size (Small, Medium, Large)
- Toppings (Cheese, Veggies, Paneer, etc.)

```
my_pizza.size        # "Medium"
my_pizza.toppings    # ["Cheese", "Olives"]
```

Methods → Actions on Pizza

These are the **things you can do** with a pizza:

- bake()
- slice()
- serve()

They are written as functions inside the class:

```
def bake(self):
    print(f"Baking a {self.size} pizza with {', '.join(self.toppings)}.")

def serve(self):
    print("Pizza is ready to serve!")
```

Constructor **init**() → Making the Pizza

When you call the recipe with your own size and toppings, Python uses the **__init__()** method to create a fresh pizza object.

```
def __init__(self, size, toppings):
    self.size = size
    self.toppings = toppings
```

self → The current pizza you're working on

Inside the class, self refers to the pizza being made. It helps keep track of which object you're working with.

Summary

OOP Concept	Pizza Example	Python Code
Class	Pizza Recipe	`class Pizza:`
Object	Real Pizza made from recipe	`my_pizza = Pizza(...)`
Attributes	Size, Toppings	`self.size`, `self.toppings`
Methods	Bake, Slice, Serve	`def bake(self): ...`
Constructor	Making the pizza	`def __init__(self): ...`
`self`	The pizza being made or used	`self.size`, `self.bake()`

Class → Blueprint

Think of a **Pizza Recipe** as a class. It's just the **instructions**, not a real pizza.

```
class Pizza:
    def __init__(self, size, toppings):
        self.size = size
        self.toppings = toppings

    def bake(self):
        print(f"Baking a {self.size} pizza with {',
'.join(self.toppings)}.")

    def serve(self):
        print("Pizza is ready to serve!")
```

In [39]:
```
"""
Problem 1: Create a Pizza class with below details

Member Attributes:
1. Size
2. Toppings

Member Functions:
1. bake()
2. Serve()

"""
class Pizza:
    def __init__(self, size, toppings):
        self.size = size
        self.toppings = toppings

    def __str__(self):
        return f"{self.size} Size Pizza with Toppings {",".join(self.toppings)}"

    def bake(self):
        print(f"Baking a {self.size} pizza with {', '.join(self.toppings)}.")
```

```
    def serve(self):
        print("Pizza is ready to serve!")
```

In [40]:
```
"""
Problem 2: You have a pizza shop. You have received 2 pizza orders You have to prep

Customer 1 (Ramesh):
- Large Pizza
- Toppings : Cheese, Panner, Capsicum

Customer 2 (Mahesh):
- Small Pizza
- Toppings : Mushrrom, Olives

"""
# Creating pizza objects
pizza_for_ramesh = Pizza("Large", ["Cheese", "Paneer", "Capsicum"])
pizza_for_ramesh.bake()
pizza_for_ramesh.serve()


pizza_for_mahesh = Pizza("Small", ["Mushroom", "Olives"])
pizza_for_mahesh.bake()
pizza_for_mahesh.serve()
```

```
Baking a Large pizza with Cheese, Paneer, Capsicum.
Pizza is ready to serve!
Baking a Small pizza with Mushroom, Olives.
Pizza is ready to serve!
```

In [41]:
```
"""
Problem 3: You have a pizza shop. You have received 20 pizza party orders You have

requests_pizza_by_ramesh =
[
    [4,"Large", ["Cheese", "Paneer", "Capsicum"]],
    [1,"Medium", ["Mushroom", "Olives"]],
    [2,"Small", ["Cheese", "Paneer", "Capsicum"]],
    [3,"Small", ["Mushroom", "Olives"]],
]

"""


requests_pizza_by_ramesh = [
    [4,"Large", ["Cheese", "Paneer", "Capsicum"]],
    [1,"Medium", ["Mushroom", "Olives"]],
    [2,"Small", ["Cheese", "Paneer", "Capsicum"]],
    [3,"Small", ["Mushroom", "Olives"]],
]

pizza_of_ramesh = []

for req in requests_pizza_by_ramesh:
  for idx in range(req[0]):
    pizza_obj = Pizza(req[1], req[2])
```

```
    pizza_obj.bake()
    pizza_obj.serve()
    pizza_of_ramesh.append(pizza_obj)
```

```
Baking a Large pizza with Cheese, Paneer, Capsicum.
Pizza is ready to serve!
Baking a Large pizza with Cheese, Paneer, Capsicum.
Pizza is ready to serve!
Baking a Large pizza with Cheese, Paneer, Capsicum.
Pizza is ready to serve!
Baking a Large pizza with Cheese, Paneer, Capsicum.
Pizza is ready to serve!
Baking a Medium pizza with Mushroom, Olives.
Pizza is ready to serve!
Baking a Small pizza with Cheese, Paneer, Capsicum.
Pizza is ready to serve!
Baking a Small pizza with Cheese, Paneer, Capsicum.
Pizza is ready to serve!
Baking a Small pizza with Mushroom, Olives.
Pizza is ready to serve!
Baking a Small pizza with Mushroom, Olives.
Pizza is ready to serve!
Baking a Small pizza with Mushroom, Olives.
Pizza is ready to serve!
```

In [42]:
```
for item in pizza_of_ramesh:
    print(item)
```

```
Large Size Pizza with Toppings Cheese,Paneer,Capsicum
Large Size Pizza with Toppings Cheese,Paneer,Capsicum
Large Size Pizza with Toppings Cheese,Paneer,Capsicum
Large Size Pizza with Toppings Cheese,Paneer,Capsicum
Medium Size Pizza with Toppings Mushroom,Olives
Small Size Pizza with Toppings Cheese,Paneer,Capsicum
Small Size Pizza with Toppings Cheese,Paneer,Capsicum
Small Size Pizza with Toppings Mushroom,Olives
Small Size Pizza with Toppings Mushroom,Olives
Small Size Pizza with Toppings Mushroom,Olives
```

Lecture 15 : Pillars of OOPs - Overview

Object-Oriented Programming (OOP) is built on four main principles called the **4 pillars.** These help us write clean, organized, and reusable code.

1. Inheritance

Inheritance means a class (child) can inherit properties and methods from another class (parent), reducing code repetition.

One Liner: One class inherits (reuses) the attributes and methods of another.

Example: Listing of Products in E-Commerce Store

In [4]:
```python
# Single Inheritance Example
# Base class
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def get_details(self):
        return f"{self.name}: ₹{self.price}"

# Subclass 1: Electronics
class Electronics(Product):
    def __init__(self, name, price, warranty):
        super().__init__(name, price) # We can access methods of parent class with
        self.warranty = warranty

    def get_details(self):
        return super().get_details() + f", Warranty: {self.warranty} years"

# Subclass 2: Clothing
class Clothing(Product):
    def __init__(self, name, price, size):
        super().__init__(name, price)
        self.size = size

    def get_details(self):
        return super().get_details() + f", Size: {self.size}"
```

In [7]:
```python
# 🍼 Creating Instances
phone = Electronics("Smartphone", 25000, 24)
tshirt = Clothing("T-Shirt", 999, "L")

# 📄 Output
print(phone.get_details())    # Product: Smartphone, Price: ₹25000, Warranty: 24 mo
print(tshirt.get_details())   # Product: T-Shirt, Price: ₹999, Size: L
```

```
Smartphone: ₹25000, Warranty: 24 years
T-Shirt: ₹999, Size: L
```

NOTE

- Electronics and Clothing inherit from Product, but extend the functionality.
- Why it's useful: Reduces code duplication and builds a clean product hierarchy.

2. Encapsulation

Encapsulation is the **bundling** of **attributes** and **methods (functions)** within a class, **restricting access** to some components to control interactions.

A class is an example of encapsulation as it encapsulates all the data (attributes) and behavior (methods) together within a single unit.

It allows hiding internal data and only allowing access through controlled methods (Getters and Setters).

One Liner: Wrapping data (variables) and code (methods) together as a single unit (class).

Example: Protecting user data or product info

In [9]:
```
class User:
    def __init__(self, username, password):
        self.__username = username         # private attribute
        self.__password = password

    def login(self, entered_password):
        if self.__password == entered_password:
            return f"Welcome {self.__username}!"
        else:
            return "Incorrect password."
```

In [12]:
```
# Usage
user = User("Sanjeev", "iamgreat@123")
# print(user.__password)  # ❌ Will raise an error: AttributeError

print(user.login("iamgreat@123"))  # ✅ Welcome Sanjeev!
print(user.login("wrong"))      # ❌ Incorrect password.
```

```
Welcome Sanjeev!
Incorrect password.
```

NOTE

- Here, `__username` and `__password` are hidden from direct access. They're only accessible through methods like `login()`.
- Why it's useful: Prevents tampering with sensitive data like passwords or prices.

3. Polymorphism

One Liner: Polymorphism means the same method name behaves differently based on the object/class using it.

Example: Showing product details on a webpage

In [15]:
```
# Base class
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def get_details(self):
        return f"{self.name}: ₹{self.price}"

# Subclass 1: Electronics
```

```
class Electronics(Product):
    def __init__(self, name, price, warranty):
        super().__init__(name, price) # We can access methods of parent class with
        self.warranty = warranty

    def get_details(self):
        return super().get_details() + f", Warranty: {self.warranty} years"

# Subclass 2: Clothing
class Clothing(Product):
    def __init__(self, name, price, size):
        super().__init__(name, price)
        self.size = size

    def get_details(self):
        return super().get_details() + f", Size: {self.size}"
```

In [14]:
```
def show_product_details(product):
    print(product.get_details())

p1 = Electronics("Laptop", 50000, 2)
p2 = Clothing("T-shirt", 599, "M")

show_product_details(p1)  # Laptop: ₹50000, Warranty: 2 years
show_product_details(p2)  # T-shirt: ₹599, Size: M
```

```
Laptop: ₹50000, Warranty: 2 years
T-shirt: ₹599, Size: M
```

NOTE

- Though both objects use `get_details()`, the output differs based on the object type.
- Why it's useful: You can treat all Product objects the same way while keeping behavior flexible.

4. Abstraction

Abstraction means hiding complex internal details and showing only essential features.

It helps focus on "what to do" rather than "how to do it."

In Python, abstraction is often done using abstract base classes (abc module).

One Liner: concept of hiding unnecessary implementation details.

Example: Payment method handling

In [18]:
```
from abc import ABC, abstractmethod

"""
PaymentMethod is an abstract class:
Inherits from ABC (Abstract Base Class).
It defines a method pay() but doesn't implchement it.
```

```
This acts like a contract: any class that inherits it must implement pay().
"""
class PaymentMethod(ABC): # Abstract class
    @abstractmethod
    def pay(self, amount):
        pass

class CreditCard(PaymentMethod): # Concrete classes.
    def pay(self, amount):
        return f"Paid ₹{amount} using Credit Card."

class UPI(PaymentMethod):
    def pay(self, amount):
        return f"Paid ₹{amount} via UPI."

# Client code
def process_payment(method: PaymentMethod, amount):
    print(method.pay(amount))
```

In [19]:
```
card = CreditCard()
upi = UPI()

process_payment(card, 500)  # Output: Paid ₹500 using Credit Card.
process_payment(upi, 750)   # Output: Paid ₹750 via UPI.
```

```
Paid ₹500 using Credit Card.
Paid ₹750 via UPI.
```

NOTE

- `PaymentMethod` is an abstract base class. You don't care how payment is processed—just that it works.
- Why it's useful: Clean interfaces, secure logic, and scalable design (e.g., you can add Wallet, COD, etc.).

Summary

OOP Pillar	Definition	E-Commerce Example
Encapsulation	Hide internal details, use methods	Hide password or cart price using private vars
Inheritance	Child class reuses parent class	`Electronics` and `Clothing` inherit `Product`
Polymorphism	Same interface, different behavior	`get_details()` works differently for each product
Abstraction	Hide complex details behind simple APIs	`PaymentMethod` defines a clean interface

Lecture 16 : Inheritance in Python

Inheritance means a class (child) can inherit properties and methods from another class (parent), reducing code repetition.

Inheritance allows you to model an **is a** relationship, where a derived class extends the functionality of a base class.

Inheritance analaogy to a human being (attributes, behaviours).

16.1 Parent and Child class

In [85]:
```
# Let's code an easy examples which shows how to create a child class and parent cl
# Base class
class Animal:
    def __init__(self, name):
        self.name = name

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

# Derived class inheriting from Animal
class Dog(Animal):
    def speak(self):
        print(f"The {self.name} barks !")
```

Dog reuses code from Animal — this is inheritance.

In [ ]:
```
# In an inheritance relationship:

# 1. Classes that inherit from another are called derived classes, subclasses, or s
# 2. Classes from which other classes are derived are called base classes or super
# 3. A derived class is said to derive, inherit, or extend a base class.
```

In [86]:
```
# Creating an instance of the derived class
dog_instance = Dog("Buddy")

# Calling methods from the base and derived classes
dog_instance.speak()  # This will call the overridden method in Dog class
```

```
The Buddy barks !
```

In [87]:
```
print(isinstance(dog_instance, Dog))      # True
print(isinstance(dog_instance, Animal))  # True
```

```
True
True
```

Method Overriding with Single Inheritance

Method overriding in Python is a mechanism that allows a subclass to provide a specific implementation for a method that is already defined in its superclass.

The overriding method in the subclass should have the same name and parameters (if overridden), but it may provide a different implementation.

In [80]:
```
# Base class
class Animal:
    def make_sound(self):
        print("Animal makes a sound")

# Subclass 1
class Dog(Animal):
    def make_sound(self):
        print("Dog barks")

# Subclass 2
class Cat(Animal):
    def make_sound(self):
        print("Cat meows")
```

In [81]:
```
animal1 = Dog()
animal2 = Cat()

animal1.make_sound()  # Calls Dog's make_sound method
animal2.make_sound()  # Calls Cat's make_sound method
```

```
Dog barks
Cat meows
```

16.2 Using super() for parent initialization

- The super() function is used to call methods in the parent class from the child class. This is particularly useful when you want to extend or modify the functionality of a parent class method, such as the **Init**() constructor method.
- So why do we use the super() function? We use the super function because we want to call and initialize the parent class's constructor and also because we want to avoid explicitly naming the parent class. This is helpful, especially in cases of multiple inheritance.

Constructor call sequence

In [80]:
```
class Vehicle:
    def __init__(self):
        print("Vehicle constructor")

    def start(self):
        print("Vehicle started")


class Car(Vehicle):
    def __init__(self):
        # The same self object is passed to both Car and Vehicle.
```

```
        # Vehicle.__init__(self) is simply initializing the same Car instance — not
        Vehicle.__init__(self) # super().__init__()
        print("Car constructor")

    def start(self):
        print("Car started")


# class ElectricCar(Car):
#     def __init__(self):
#         super().__init__()
#         print("ElectricCar constructor")

#     def start(self):
#         print("ElectricCar started")
```

In [81]:
```
c = Car()
```

```
Vehicle constructor
Car constructor
```

In [82]:
```
class Person:
    def __init__(self, name, id):
        self.name = name
        self.id = id

class Student(Person):
    def __init__(self, name, id, grade):
        # Using super() to initialize the parent class
        super().__init__(name, id)
        self.grade = grade
```

In [ ]:
```
# Example usage
student = Student("Samuel", 5678, "B+")
print(student.name)
print(student.id)
print(student.grade)
```

In [ ]:
```
# Problem Practice on Inheritance
```

In [ ]:
```
# Single Inheritance Example
# Base class
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def get_details(self):
        return f"{self.name}: ₹{self.price}"

# Subclass 1: Electronics
class Electronics(Product):
    def __init__(self, name, price, warranty):
        super().__init__(name, price) # We can access methods of parent class with
        self.warranty = warranty
```

```
    def get_details(self):
        return super().get_details() + f", Warranty: {self.warranty} years"

# Subclass 2: Clothing
class Clothing(Product):
    def __init__(self, name, price, size):
        super().__init__(name, price)
        self.size = size

    def get_details(self):
        return super().get_details() + f", Size: {self.size}"
```

In [ ]:
```
# Creating Instances
phone = Electronics("Smartphone", 25000, 24)
tshirt = Clothing("T-Shirt", 999, "L")

# Output
print(phone.get_details())    # Product: Smartphone, Price: ₹25000, Warranty: 24 mo
print(tshirt.get_details())   # Product: T-Shirt, Price: ₹999, Size: L
```

16.3 Interesting Fact about classes in python

In [92]:
```
class EmptyClass:
    pass
```

In [93]:
```
c = EmptyClass()
print(dir(c)) #  dir() function to list all the members in the specified object
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__firstlin
eno__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__has
h__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__static_attributes__', '__str__', '__subclasshook__', '__weakref__']
```

In [94]:
```
o = object()
print(dir(o))
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subcla
ss__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__re
pr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

- every single member of the object class is also present in EmptyClass.
- This is because every class that you create in Python implicitly derives from object(Topmost base class).
- You could be more explicit and write class EmptyClass(object):, but it's redundant and unnecessary.

In [96]:
```
ins = EmptyClass()

print(type(ins))        # 👉 <class '__main__.EmptyClass'> =>  ins is an instance
print(type(EmptyClass))    # 👉 <class 'type'> => EmptyClass itself is an object o
```

```
print(isinstance(ins, object))      # 👉 True => All instances inherit from the obj
print(isinstance(EmptyClass, object))  # 👉 True => Even the class itself is an obj
```

```
<class '__main__.EmptyClass'>
<class 'type'>
True
True
```

In Python, type is the metaclass — it is used to create all classes, including itself.

Let's check what it inherits from:

In [34]:
```
print(type.__bases__)   # Output: (<class 'object'>,)
```

```
(<class 'object'>,)
```

type inherits from object

In [ ]:
```
# Twist
print(type(object))     # Output: <class 'type'>
print(type(type))       # Output: <class 'type'>
```

This creates a circular but consistent system

Element	Created By	Inherits From
`object`	`type`	— (no parent)
`type`	`type`	`object`
Any user-defined class	`type`	Typically `object`

16.4 Accessibility of members in Inheritance

The **visibility of inherited members (attributes and methods)** in Python depends on their access modifiers.

In Python, access modifiers are used to control the visibility and accessibility of attributes and methods within a class.

Python does not have explicit keywords like "public," "private," or "protected".

Instead, It relies on naming conventions to indicate the intended visibility

---

1. Public (default): Members are accessible from anywhere, both within the class and outside the class.
2. Protected (_single): Members are accessible within the class, within derived classes, and within the same module. However, they are considered conventionally private, and their use outside the class or module is discouraged.

3. Private (__double): Members are accessible only within the class. They are not accessible in derived classes or outside the class.

Member Visibility	Public (default)	Protected (_single)	Private (__double)
In Base Class	Accessible	Accessible	Accessible
In Derived Class	Accessible	Accessible within subclass/module	Not Accessible

> **Python does not enforce strict access control. It relies on conventions and developer discipline.**

In [90]:
```
class Parent:
    def __init__(self):
        self.public = "Public"
        self._protected = "Protected"
        self.__private = "Private"

    def show(self):
        print(f"Inside Parent: {self.public}, {self._protected}, {self.__private}")

class Child(Parent):
    def access_members(self):
        print("Inside Child:")
        print(f"Public: {self.public}")            # ✅ Accessible
        print(f"Protected: {self._protected}")      # ✅ Accessible (by convention)
        # print(f"Private: {self.__private}")       # ❌ AttributeError
```

In [91]:
```
c = Child()
c.show()
c.access_members()
```

```
Inside Parent: Public, Protected, Private
Inside Child:
Public: Public
Protected: Protected
```

16.5 Types of Inheritance in Python

Python supports five types of inheritance:

1. Single inheritance
2. Hierarchical inheritance
3. Multilevel inheritance
4. Multiple inheritance
5. Hybrid inheritance

1. Single Inheritance - Explained with examples

Single inheritance is a type of inheritance in object-oriented programming where a class inherits from only one base class.

In [ ]:
```python
# Single Inheritance Example
# Base class
class Animal:
    def __init__(self, name):
        self.name = name

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

# Derived class inheriting from Animal
class Dog(Animal):
    def speak(self):
        print(f"The {self.name} barks!")
```

In [ ]:
```python
# Creating an instance of the derived class
dog_instance = Dog("Buddy")

# Calling methods from the base and derived classes
dog_instance.speak()  # This will call the overridden method in Dog class
```

```
Buddy says Woof!
```

2. Hierarchical inheritance - Explained with Examples

In hierarchical inheritance, a single base class (parent class) is inherited by multiple derived classes (child classes).

Each derived class shares common attributes and methods from the base class but may have its own additional attributes and methods.

In [ ]:
```python
# Hierarchical Inheritance explained with Examples
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"

class Bird(Animal):
    def speak(self):
        return f"{self.name} sings beautifully!"
```

In [ ]:
```python
# Creating objects of the derived classes
dog = Dog("Buddy")
cat = Cat("Whiskers")
bird = Bird("Tweetie")
```

```
# Calling the speak method on each object
print(dog.speak())   # Output: Buddy says Woof!
print(cat.speak())   # Output: Whiskers says Meow!
print(bird.speak())  # Output: Tweetie sings beautifully!
```

In [ ]:
```
# Problem Practice: Hierarchical Inheritance in Vehicle Classes
```

In [ ]:
```
# Base class
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start(self):
        print("Starting the", self.make, self.model)

    def stop(self):
        print("Stopping the", self.make, self.model)

# Derived class Car inheriting from Vehicle
class Car(Vehicle):
    def __init__(self, make, model, numberOfDoors):
        super().__init__(make, model)
        self.numberOfDoors = numberOfDoors

    def honk(self):
        print("Honking the horn of the", self.make, self.model)

# Derived class Motorcycle inheriting from Vehicle
class Motorcycle(Vehicle):
    def __init__(self, make, model, engineType):
        super().__init__(make, model)
        self.engineType = engineType

    def wheelie(self):
        print("Performing a wheelie on the", self.make, self.model)

# Create an instance of the Car class
my_car = Car("Toyota", "Camry", 4)
my_car.start()
my_car.honk()
my_car.stop()
```

```
Starting the Toyota Camry
Honking the horn of the Toyota Camry
Stopping the Toyota Camry
```

In [ ]:
```
# Create an instance of the Motorcycle class
my_motorcycle = Motorcycle("Harley-Davidson", "Sportster", "4-stroke")
my_motorcycle.start()
my_motorcycle.wheelie()
my_motorcycle.stop()
```

3. Multilevel Inheritance - Explained with Examples

Multilevel inheritance in Python involves creating a chain of classes where each class extends the previous one.

In other words, a derived class serves as the base class for another class.

In [ ]:
```python
# Multilevel Inheritance Examples
class Animal:
    def __init__(self, name):
        self.name = name

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

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

class Labrador(Dog):
    def swim(self):
        print(f"{self.name} can swim.")
```

In [ ]:
```python
# Creating instances of the classes
animal = Animal("Generic Animal")
dog = Dog("Buddy")
labrador = Labrador("Max")

# Calling methods
animal.speak()      # Output: Generic Animal makes a sound.
dog.speak()         # Output: Buddy makes a sound.
dog.bark()          # Output: Buddy barks.
labrador.speak()    # Output: Max makes a sound.
labrador.bark()     # Output: Max barks.
labrador.swim()     # Output: Max can swim.
```

```
Generic Animal makes a sound.
Buddy makes a sound.
Buddy barks.
Max makes a sound.
Max barks.
Max can swim.
```

Python follows **MRO (Method Resolution Order)** — it looks in the first parent listed (Camera), then Phone, then object.

You can check the **MRO**:

In [ ]:
```python
print(Smartphone.__mro__)
```

4. Multiple inheritance - Explained with examples

In [43]:
```python
class Camera:
    def take_photo(self):
        return "Taking a photo"
```

```
class Phone:
    def make_call(self, number):
        return f"Calling {number}"

# Smartphone inherits from both Camera and Phone
class Smartphone(Camera, Phone):
    def browse_internet(self):
        return "Browsing the internet"
```

In [44]:
```
s = Smartphone()

print(s.take_photo())           # from Camera
print(s.make_call("9876543210")) # from Phone
print(s.browse_internet())      # from Smartphone
```

```
Taking a photo
Calling 9876543210
Browsing the internet
```

5. Hybrid inheritance - Explained with examples

In [70]:
```
class Person:
    def __init__(self, name, **kwargs):
        self.name = name
        print(f"Person __init__ called for {self.name}")
        super().__init__(**kwargs)

class Student(Person):
    def __init__(self, student_id, **kwargs):
        self.student_id = student_id
        print(f"Student __init__ called for {kwargs.get('name')}")
        super().__init__(**kwargs)

    def study(self):
        return f"{self.name} is studying."

class Teacher(Person):
    def __init__(self, subject, **kwargs):
        self.subject = subject
        print(f"Teacher __init__ called for {kwargs.get('name')}")
        super().__init__(**kwargs)

    def teach(self):
        return f"{self.name} teaches {self.subject}."

class TeachingAssistant(Student, Teacher):
    def __init__(self, name, student_id, subject):
        super().__init__(name=name, student_id=student_id, subject=subject)
        print(f"TeachingAssistant __init__ called for {self.name}")

    def assist(self):
        return f"{self.name} is assisting in {self.subject}."
```

In [72]:
```
ta = TeachingAssistant("Ravi", "S123", "Math")
print(ta.study())
```

```
print(ta.teach())
print(ta.assist())
```

```
Student __init__ called for Ravi
Teacher __init__ called for Ravi
Person __init__ called for Ravi
TeachingAssistant __init__ called for Ravi
Ravi is studying.
Ravi teaches Math.
Ravi is assisting in Math.
```

16.6 Benefits and Limitations of inheritance

Benefits

- **Reusability**: With inheritance you can write code once in the parent class and reuse it in the child classes. Using the example, both FullTimeEmployee and Contractor can inherit a get_details() method from the Employee parent class.
- **Simplicity**: Inheritance models relationships clearly. A good example is the FullTimeEmployee class which “is-a” type of the Employee parent class.
- **Scalability**: It also add new features or child classes without affecting existing code. For example, we can easily add a new Intern class as a child class.

Limitations

- **Complexity**: This won't be surprising, but too many levels of inheritance can make the code hard to follow. For example, if an Employee has too many child classes like Manager, Engineer, Intern, etc., it may become confusing.
- **Dependency**: Changes to a parent class can unintentionally affect all subclasses. If you modify Employee for example, it might break FullTimeEmployee or Contractor.
- **Misuse**: Using inheritance when it is not the best fit can complicate designs. You would not want to create a solution where Car inherits from Boat just to reuse move(). The relationship doesn’t make sense.

Lecture 17 : Encapsulation in Python

Encapsulation is the **bundling** of **data (attributes)** and **methods (functions)** within a class, **restricting access** to some components to control interactions.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

It allows hiding internal data and only allowing access through controlled methods.

One Liner: Wrapping data (variables) and code (methods) together as a single unit (class).

Abstractions Vs Encapsulation

Feature	Encapsulation	Abstraction
Purpose	Hide internal state/data	Hide implementation complexity
What is hidden?	Data (e.g., password, cart total)	Logic/Details (e.g., how `.pay()` works)
How?	Private variables (`__var`) + methods	Abstract classes, interfaces, clean APIs
User sees	Controlled access to variables	Only the interface (`pay` , `refund` , etc.)
Example	`__password` , `__total_price`	`PaymentMethod.pay()`
Goal	Protect data from misuse	Simplify usage of complex systems

Example:

In [60]:
```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance  # 👈 Private variable (encapsulated)

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

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew ₹{amount}")
        else:
            print("Insufficient balance or invalid amount.")

    def get_balance(self):
        return self.__balance  # 👈 Controlled access
```

In [61]:
```
# Usage
account = BankAccount("Sanjeev", 1000)
account.deposit(500)
account.withdraw(200)

print("Current Balance:", account.get_balance())
```

```
Deposited ₹500
Withdrew ₹200
Current Balance: 1300
```

In [62]:
```python
# Trying to access private variable directly (not allowed)
# print(account.__balance)  ❌ This will cause an error
```

Access Modifiers in Python :

In Python, access modifiers are used to control the visibility and accessibility of attributes and methods within a class.

Python does not have explicit keywords like "public," "private," or "protected".

Instead, It relies on naming conventions to indicate the intended visibility

1.**Public**: By default, all attributes and methods in a class are considered public. They can be accessed and modified from outside the class.

2.**Protected**: Attributes and methods intended for internal use within the class and its subclasses are often marked as protected by prefixing them with a single underscore (_).

3.**Private**: Attributes and methods that should not be accessed from outside the class are conventionally marked as private by prefixing them with a double underscore (__).

In [ ]:
```python
class Person:
    def __init__(self):
        self.name = "Alice"          # public attribute
        self._age = 30               # protected attribute
        self.__salary = 50000        # private attribute

    def display(self):
        print("Name:", self.name)
        print("Age:", self._age)
        print("Salary:", self.__salary)
```

In [ ]:
```python
obj = Person()
obj.display()

# Accessing attributes from outside
print(obj.name)          # ✅ Public: Accessible
print(obj._age)          # ⚠️ Protected: Accessible but discouraged
print(obj.__salary)      # ❌ Private: Will raise AttributeError
```

In [ ]:
```python
# Accessing Private Variables (Name Mangling)
obj._Person__salary
```

Out[ ]:
```
50000
```

> Python does not enforce strict access control. It relies on conventions and developer discipline.

Accessing and Modifying Private Data Members:

In Python, getter and setter methods are used to access and modify private data members (fields) of a class.

In [84]:
```
class MyClass :
    def __init__(self):
        __myField = None

    # Getter method for myField
    def getMyField(self):
        return MyClass.__myField

    # Setter method for myField
    def setMyField(self , value):
        MyClass.__myField = value
```

Lecture 18 : Polymorphism in Python

Polymorphism in Python

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different types to be treated as objects of a common base type.

It enables flexibility in code design and promotes code reuse. Here are the two main types of polymorphism in Python:

In [10]:
```
# operator polymorphism - Operator polymorphism, or operator overloading, means tha

int1 = 10
int2 = 15
print(int1 + int2)

str1 = "10"
str2 = "15"
print(str1 + str2)

# Function Polymorphism -  len() function, for instance, can be used to return the
# However, it will measure the length of the object differently depending on the ob

str1 = "animal"
print(len(str1))
# returns 6

list1 = ["giraffe","lion","bear","dog"]
print(len(list1))
# returns 4
```

```
25
1015
6
4
```

In [ ]:
```python
# Object polymorphism

class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):
    def speak(self):
        return "Bark"

class Cat(Animal):
    def speak(self):
        return "Meow"

def make_sound(animal):
    print(animal.speak())

# Object Polymorphism in action
animals = [Dog(), Cat()]
for a in animals:
    make_sound(a)

#Even though Dog and Cat are different objects, they are treated as Animal and used
```

Method Overloading:

In some languages, such as Java or C++, method overloading allows you to define multiple methods with the same name in the same class, but with different parameter lists.

In Python, method overloading is achieved in a different way, as the language does not support multiple methods with the same name but different parameter lists.

In [ ]:
```python
# If you define multiple methods with the same name, only the last one will be used
```

In [ ]:
```python
class MyClass:
    def add(self, a, b):
        return a + b

    def add(self, a, b, c):
        return a + b + c

# So in your code, def add(self, a, b) is overwritten by def add(self, a, b, c).
# This will result in an error in Python
```

In [ ]:
```python
obj = MyClass()
print(obj.add(1, 2))     # ❌ TypeError: add() missing 1 required positional argume
print(obj.add(1, 2, 3))  # ✅ Works fine: returns 6
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-22-1720559205.py in <cell line: 0>()
      1 obj = MyClass()
----> 2 print(obj.add(1, 2))     # ❌ TypeError: add() missing 1 required positiona
l argument: 'c'
      3 print(obj.add(1, 2, 3))  # ✅ Works fine: returns 6

TypeError: MyClass.add() missing 1 required positional argument: 'c'
```

Instead of method overloading, Python uses a single method with optional or default parameters to achieve similar functionality.

In [ ]:
```python
# default parameters

class MyClass:
    def add(self, a, b=0, c=0):
        return a + b + c

# Creating an instance of MyClass
my_object = MyClass()

# Calling the add method with different parameter lists
result1 = my_object.add(1)
result2 = my_object.add(1, 2)
result3 = my_object.add(1, 2, 3)

print(result1)  # Output: 1
print(result2)  # Output: 3
print(result3)  # Output: 6
```

```
1
3
6
```

In [ ]:
```python
# Using Variable-Length Argument Lists

class MyClass:
    def add(self, *args):
        return sum(args)

# Creating an instance of MyClass
my_object = MyClass()

# Calling the add method with different numbers of arguments
result1 = my_object.add(1)
result2 = my_object.add(1, 2)
result3 = my_object.add(1, 2, 3)

print(result1)  # Output: 1
print(result2)  # Output: 3
print(result3)  # Output: 6
```

This is how python achieves compile-time polymorphism. If we pass 2 arguments, the value of c will be set to the default value provided. Otherwise, it will be set to the passed value.

Operator Overloading in Python is supported using special methods `(__add__, __len__, etc.)`.

Function Overloading isn't natively supported — if you define a function twice, the last one overrides the previous.

In [ ]:
```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

# Usage
v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2  # calls __add__, operator overloading
```

In [12]:
```python
# NOTE:
# All overloading and polymorphism in Python are resolved at run-time, after byteco
# Python is a dynamically typed language
# Dynamic (late) binding is used, not static (early) binding
# Even operator like + becomes __add__() at run-time, not at parse-time or compile-
```

Method Overriding:

Method overriding is a form of polymorphism that occurs at runtime.

In Python, a subclass can provide a specific implementation for a method that is already defined in its superclass.

This allows objects of the derived class to be used interchangeably with objects of the base class.

In [ ]:
```python
class Animal:
    def make_sound(self):
        return "Some generic sound"

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Example usage of method overriding
dog = Dog()
```

```
cat = Cat()

print(dog.make_sound())  # Output: Woof!
print(cat.make_sound())  # Output: Meow!
```

```
Woof!
Meow!
```

Method overriding is a concept in object-oriented programming where a subclass provides a specific implementation for a method that is already defined in its superclass.

This allows the subclass to provide a specialized behavior while still maintaining the same method signature as the superclass.

In Python, method overriding is achieved by creating a method in the subclass with the same name as the method in the superclass.

Key Points

1. Method overriding is a form of run-time polymorphism, and the specific implementation to be called is determined at runtime based on the type of the object.
2. The `super()` function can be used to call the overridden method from the superclass within the overridden method of the subclass if needed.

Duck Typing

Duck typing is a concept in Python where the type of an object is less important than the methods and properties it has.

“If it walks like a duck and quacks like a duck, it's probably a duck.”

So in Python, if an object has the right methods or attributes, you can use it, regardless of its actual class.

In [ ]:
```
class CreditCard:
    def pay(self, amount):
        print(f"Paid ₹{amount} using Credit Card.")

class UPI:
    def pay(self, amount):
        print(f"Paid ₹{amount} using UPI.")

class PayPal:
    def pay(self, amount):
        print(f"Paid ₹{amount} using PayPal.")

# Duck typing function
def process_payment(payment_method, amount):
    payment_method.pay(amount)  # No type check!

# Different objects with the same method name
```

```
process_payment(CreditCard(), 500)
process_payment(UPI(), 200)
process_payment(PayPal(), 1000)
```

In [ ]:
```
Why It's Duck Typing:
The function process_payment() doesn't care what type of object it receives.
It only cares that the object has a .pay() method.
This is "If it quacks like a duck…" — behavior-based programming, not type-based.
```

Example: E-Commerce Product

In [4]:
```
# Step 1: Create a Base Class Product
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def get_details(self):
        return f"{self.name}: ₹{self.price}"
```

In [5]:
```
# Step 2: Create Subclass Electronics
class Electronics(Product):
    def __init__(self, name, price, warranty):
        super().__init__(name, price)
        self.warranty = warranty

    def get_details(self):
        return super().get_details() + f", Warranty: {self.warranty} years"

# This class overrides the get_details() method — adds warranty info to the base cl
```

In [6]:
```
# Step 3: Create Subclass Clothing
class Clothing(Product):
    def __init__(self, name, price, size):
        super().__init__(name, price)
        self.size = size

    def get_details(self):
        return super().get_details() + f", Size: {self.size}"
#  Again, get_details() is overridden to include size for clothing.
```

In [7]:
```
# Step 4: Use Polymorphism in Action :
# Polymorphism in action
products = [
    Electronics("Smartphone", 25000, 2),
    Clothing("T-shirt", 999, "M"),
    Product("Book", 500)
]

for item in products:
    print(item.get_details())
```

```
Smartphone: ₹25000, Warranty: 2 years
T-shirt: ₹999, Size: M
Book: ₹500
```

In [ ]:
```
# What Happened Here?
# You called the same method get_details() on different objects.
# Each object responded differently, based on its class.
# This is runtime polymorphism — the method that gets executed is determined at run
```

Lecture 19: Abstraction in Python

19.1 Abstract Class in Python

In [ ]:
```
till now we have seen concrete class
what to do not how to do
```

One-Liner: An abstract class is like a template for other classes.

It defines methods that must be included in any class that inherits from it, but it doesn't provide the actual code for those methods.

An abstract class in Python is a class that cannot be instantiated and is meant to be inherited by other classes. It can define abstract methods (methods with no implementation) that must be implemented by any subclass.

Usecase: Abstract classes are used to define a common interface for a group of related classes.

- Abstract classes are defined using the `abc` module.
- Use `@abstractmethod` decorator to mark methods that must be overridden.
- You cannot create objects of an abstract class directly.

Creating a Python Abstract Class

- The ABC class is a built-in Python feature that serves as a fundamental basis for developing abstract classes. You must inherit from ABC to define an abstract class. The class is abstract and cannot be instantiated directly, as indicated by this inheritance.
- You can define abstract methods inside an abstract class by using the abstractmethod decorator.
- Any concrete subclass must implement and override abstract methods, which are placeholders.
- This makes the code consistent and predictable by guaranteeing that all derived classes offer functionality for the designated methods.

In [25]:
```
from abc import ABC, abstractmethod

class Shape(ABC):
```

```
    @abstractmethod
    def area(self):
        pass
```

In [23]:
```
shape = Shape() # error
```

In [13]:
```
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
```

In [14]:
```
#circle = Circle() #error
```

In [15]:
```
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2
```

In [16]:
```
circle = Circle(5)
print(f"Area: {circle.area()}")
```

```
Area: 78.53975
```

In [20]:
```
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    def concrete_method(self):
        print("concrete method in abstract class")

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2

    def concrete_method(self):
        print("concrete method in concrete class")
```

In [21]:
```
circle = Circle(5)
circle.concrete_method()
```

```
concrete method in concrete class
```

Types of Abstraction

- Partial Abstraction: Abstract class contains both abstract and concrete methods.
- Full Abstraction: Abstract class contains only abstract methods (like interfaces).

Why use abstract classes in Python?

- Enforce method implementation: An abstract class's methods function as a contract, requiring each subclass to supply its own implementation.
- Encourage code reuse: abstract classes can have both concrete and abstract methods.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

19.2 Abstraction in Python

Abstraction means hiding complex internal details and showing only essential features.

It helps focus on "what to do" rather than "how to do it."

One Liner: concept of hiding unnecessary implementation details

In Python, abstraction is often done using abstract base classes (abc module).

Example:

In [ ]:

In [18]:
```python
from abc import ABC, abstractmethod

"""
PaymentMethod is an abstract class:
Inherits from ABC (Abstract Base Class).
It defines a method pay() but doesn't implchement it.
This acts like a contract: any class that inherits it must implement pay().
"""
class PaymentMethod(ABC): # Abstract class
    @abstractmethod
    def pay(self, amount):
        pass

class CreditCard(PaymentMethod): # Concrete classes.
```

```
    def pay(self, amount):
        return f"Paid ₹{amount} using Credit Card."

class UPI(PaymentMethod):
    def pay(self, amount):
        return f"Paid ₹{amount} via UPI."

# Client code
def process_payment(method: PaymentMethod, amount):
    print(method.pay(amount))
```

In [19]:
```
card = CreditCard()
upi = UPI()

process_payment(card, 500)  # Output: Paid ₹500 using Credit Card.
process_payment(upi, 750)   # Output: Paid ₹750 via UPI.
```

```
Paid ₹500 using Credit Card.
Paid ₹750 via UPI.
```

NOTE

- `PaymentMethod` is an abstract base class. You don't care how payment is processed—just that it works.
- Why it's useful: Clean interfaces, secure logic, and scalable design (e.g., you can add Wallet, COD, etc.).

Abstractions Vs Encapsulation

Feature	Encapsulation	Abstraction
Purpose	Hide internal state/data	Hide implementation complexity
What is hidden?	Data (e.g., password, cart total)	Logic/Details (e.g., how `.pay()` works)
How?	Private variables (`__var`) + methods	Abstract classes, interfaces, clean APIs
User sees	Controlled access to variables	Only the interface (`pay` , `refund` , etc.)
Example	`__password` , `__total_price`	`PaymentMethod.pay()`
Goal	Protect data from misuse	Simplify usage of complex systems

key Points :

1. Abstract classes cannot be instantiated directly.
2. Abstract methods are declared using the `@abstractmethod` decorator in the abstract class.

3. Subclasses must provide concrete implementations for all abstract methods to be considered valid.

4. Abstract classes can contain both abstract and non-abstract methods.

Why to use Abstraction ?

Abstraction ensures consistency in derived classes by enforcing the implementation of abstract methods.

References

1. https://www.codechef.com/learn/course/oops-concepts-in-python
2. https://realpython.com/python3-object-oriented-programming/
3. https://www.sanfoundry.com/object-oriented-programming-oop-in-python/
4. https://www.geeksforgeeks.org/python/python-oops-concepts/