RESEARCH INNOVATION DISCOVERY

RID संस्था समस्या का समाधान

RUN BY TWKSAA WELFARE FOUNDATION

Reg.No: 048884
Foundation Day
05-09-2024

Advance Python

E-Book

Er. Rajesh Prasad(B.E, M.E)
Founder: RID Organization

- **RID ORGANIZATION** यानि **Research, Innovation and Discovery** संस्था जिसका मुख्य उदेश्य हैं आने वाले समय में सबसे पहले **NEW (RID, PMS & TLR)** की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से भारतीय संस्कृति, सभ्यता एवं भाषा में ही हो |
- देश, समाज, एवं लोगों की समस्याओं का समाधान **NEW (RID, PMS & TLR)** के माध्यम से किया जाये इसके लिए ही मैं राजेश प्रसाद **इस RID संस्था** की स्थपना किया हूँ|
- Research, Innovation & Discovery में रूचि रखने वाले आप सभी विधार्थियों, शिक्षकों एवं बुधीजिवियो से मैं आवाहनं करता हूँ की आप सभी **इस RID संस्था** से जुड़ें एवं अपने बुध्दि, विवेक एवं प्रतिभा से दुनियां को कुछ नई **(RID, PMS & TLR)** की खोजकर, बनाकर एवं अपनाकर लोगों की समस्याओं का समाधान करें |

"त्वक्सा एडवांस पाइथन के इस ई-पुस्तक में आप पाइथन से जुड़ी सभी बुनियादी अवधारणाएँ सीखेंगे। । मुझे आशा है कि इस ई-पुस्तक को पढ़ने के बाद आपके ज्ञान में वृद्धि होगी और आपको कंप्यूटर विज्ञान के बारे में और अधिक जानने में रुचि होगी"

"In this E-book of TWKSAA Advance Python you will learn all the basic concepts related to python. I hope after reading this book your knowledge will be improve and you will get more interest to know more thing about computer Science".

Research करने के लिए E-Mail करें:
ridorg.in@gmail.com

Research करने के लिए Message करें:
9202707903

RID हमें क्यों करना चाहिए

(Research)

अनुसंधान हमें क्यों करना चाहिए ?

Why should we do research?

1. नई ज्ञान की प्राप्ति (Acquisition of new knowledge)
2. समस्याओं का समाधान(To Solving problems)
3. सामाजिक प्रगति (To Social progress)
4. विकास को बढ़ावा देने(To promote development)
5. तकनीकी और व्यापार में उन्नति(To advances in technology & business)
6. देश विज्ञान और प्रौद्योगिकी के विकास(To develop the country's science & technology)

(Innovation)

नवीनीकरण हमें क्यों करना चाहिए ?

Why should we do Innovation?

1. प्रगति के लिए(To progress)
2. परिवर्तन के लिए(For change)
3. उत्पादन में सुधार(To Improvement in production)
4. समाज को लाभ(To Benefit to society)
5. प्रतिस्पर्धा में अग्रणी (To be ahead of competition)
6. देश विज्ञान और प्रौद्योगिकी के विकास। (To develop the country's science & technology)

(Discovery)

खोज हमें क्यों करना चाहिए?

Why should we do Discovery?

1. नए ज्ञान की प्राप्ति(Acquisition of new knowledge)
2. अविष्कारों की खोज(To Discovery of inventions)
3. समस्याओं का समाधान(To Solving problems)
4. ज्ञान के विकास में योगदान(Contribution to development of knowledge)
5. समाज के उन्नति के लिए (for progress of society)
6. देश विज्ञान और तकनीक के विकास (To develop the country's science & technology)

New Technology पर Research करने के लिए सम्पर्क करें: ridorg.in@gmail.com Mob.No: 9202707903

❖ Research(अनुसंधान):

- अनुसंधान एक प्रणालीकरण कार्य होता है जिसमें विशेष विषय या विषय की नई ज्ञान एवं समझ को प्राप्त करने के लिए सिद्धांतिक जांच और अध्ययन किया जाता है। इसकी प्रक्रिया में डेटा का संग्रह और विश्लेषण, निष्कर्ष निकालना और विशेष क्षेत्र में मौजूदा ज्ञान में योगदान किया जाता है। अनुसंधान के माध्यम से विज्ञान, प्रोधोगिकी, चिकित्सा, सामाजिक विज्ञान, मानविकी, और अन्य क्षेत्रों में विकास किया जाता है। अनुसंधान की प्रक्रिया में अनुसंधान प्रश्न या कल्पनाएँ तैयार की जाती हैं, एक अनुसंधान योजना डिज़ाइन की जाती है, डेटा का संग्रह किया जाता है, विश्लेषण किया जाता है, निष्कर्ष निकाला जाता है और परिणामों को उचित दर्शाने के लिए समाप्ति तक पहुंचाया जाता है।

❖ Innovation(नवीनीकरण): -

- Innovation एक विशेषता या नई विचारधारा की उत्पत्ति या नवीनीकरण है। यह नए और आधुनिक विचारों, तकनीकों, उत्पादों, प्रक्रियाओं, सेवाओं या संगठनात्मक ढंगों का सृजन करने की प्रक्रिया है जिससे समस्याओं का समाधान, प्रतिस्पर्धा में अग्रणी होने, और उपयोगकर्ताओं के अनुकूलता में सुधार किया जा सकता है।

❖ Discovery (आविष्कार):

- Discovery का अर्थ होता है "खोज" या "आविष्कार"। यह एक विशेषता है जो किसी नए ज्ञान, अविष्कार, या तत्व की खोज करने की प्रक्रिया को संदर्भित करता है। खोज विज्ञान, इतिहास, भूगोल, तकनीक, या किसी अन्य क्षेत्र में हो सकती है। इस प्रक्रिया में, व्यक्ति या समूह नए और अज्ञात ज्ञान को खोजकर समझने का प्रयास करते हैं और इससे मानव सभ्यता और विज्ञान-तकनीकी के विकास में योगदान देते हैं।

नोट : अनुसंधान विशेषता या विषय पर नई ज्ञान के प्राप्ति के लिए सिस्टमैटिक अध्ययन है, जबकि आविष्कार नए और अज्ञात ज्ञान की खोज है।

सुविचार:

1.	**समस्याओं का समाधान करने का उतम मार्ग हैं।** → शिक्षा ,RID, प्रतिभा, सहयोग, एकता एवं समाजिक-कार्य
2.	**एक इंसान के लिए जरूरी हैं।** → रोटी, कपड़ा, मकान, शिक्षा, रोजगार, इज्जत और सम्मान
3.	**एक देश के लिए जरूरी हैं।** - → संस्कृति-सभ्यता, भाषा, एकता, आजादी, संविधान एवं अखंडता
4.	**सफलता पाने के लिए होना चाहिए।** → लक्ष्य, त्याग, इच्छा-शक्ति, प्रतिबद्धता, प्रतिभा, एवं सतता
5.	**मरने के बाद इंसान छोडकर जाता हैं।** → शरीर, अन-धन, घर-परिवार, नाम, कर्म एवं विचार
	शरीर को बदनाम मत करों, अन-धन, पर अहंकार मत करों, घर-परिवार से प्यार करों, नाम के लिए अच्छे कर्म करो, सही विचार दो एवं पालन करों
6.	**मरने के बाद इंसान को इस धरती पर याद किया जाता हैं उनके**
	➔ नाम, काम, दान, विचार, सेवा-समपर्ण एवं कर्मो से.....

आशीर्वाद (बड़े भैया जी)

Mr. RAMASHANKAR KUMAR

मार्गदर्शन एवं सहयोग

Mr. GAUTAM KUMAR

... सोच है जिनकी नई कुछ कर दिखाने की, खोज है रीड संस्था को उन सभी इंसानों की...

"अगर आप भी **Research, Innovation and Discovery** के क्षेत्र में रूचि रखतें हैं एवं अपनी प्रतिभा से दुनियां को कुछ नया देना चाहतें एवं अपनी समस्या का समाधान **RID** के माध्यम से करना चाहतें हैं तो **RID ORGANIZATION (रीड संस्था)** से जरुर जुड़ें" || धन्यवाद || **Er. Rajesh Prasad (B.E, M.E)**

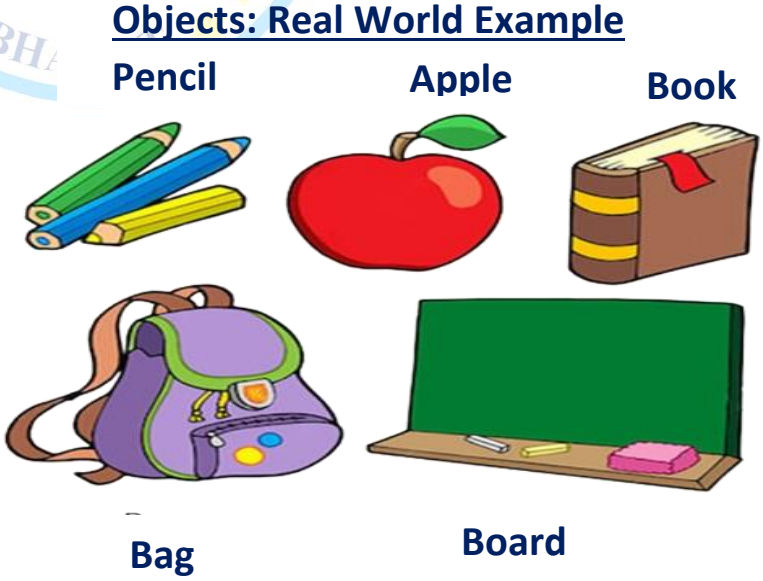
OOPS CONCEPTS
Object Oriented Programming System

- Object-Oriented Programming is a methodology or paradigm to design a program using **classes** and **objects.**
- The programming paradigm where everything is represented as an **object** is known as truly **object-oriented programming** language. Smalltalk is considered as the first truly object-oriented programming language.
- It simplifies the software development and maintenance by providing following concepts.
    1. Object
    2. Class
    3. Inheritance
    4. Polymorphism
    5. Abstraction
    6. Encapsulation

1. Object:

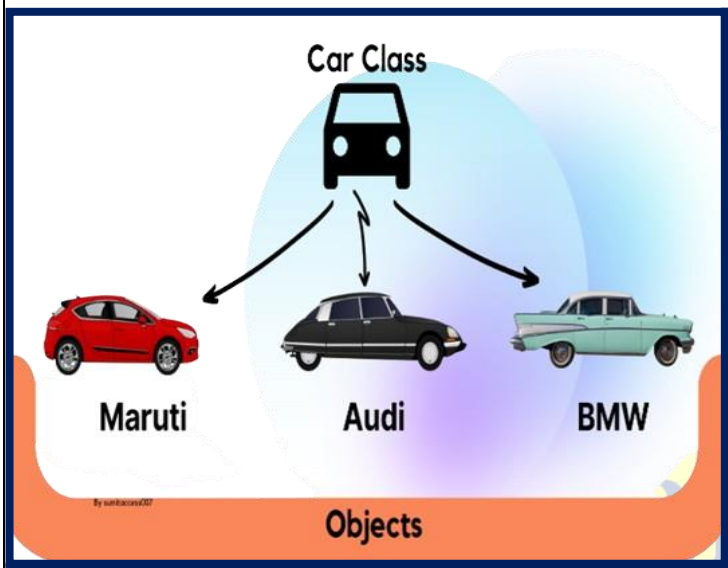
- Object means a real word entity such as pen, chair, table etc.
- Any entity that has **state** and **behavior** is known as an object.
- For example: chair, pen, table, keyboard, bike etc. It can be **physical** and **logical.**
- **State** tells us how the object looks or what properties it has. **Behavior** tells us what the object does.
- object is a specific **instance** of a class.
- **instance** is a specific realization of any objects.

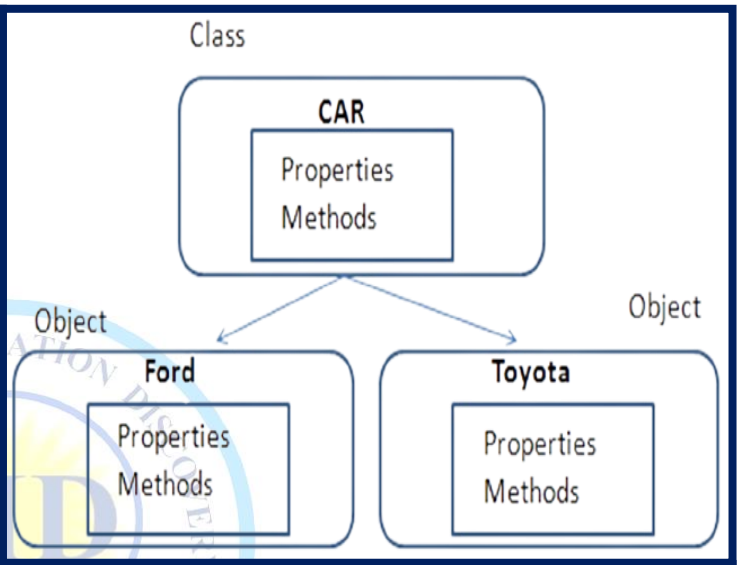
Example:

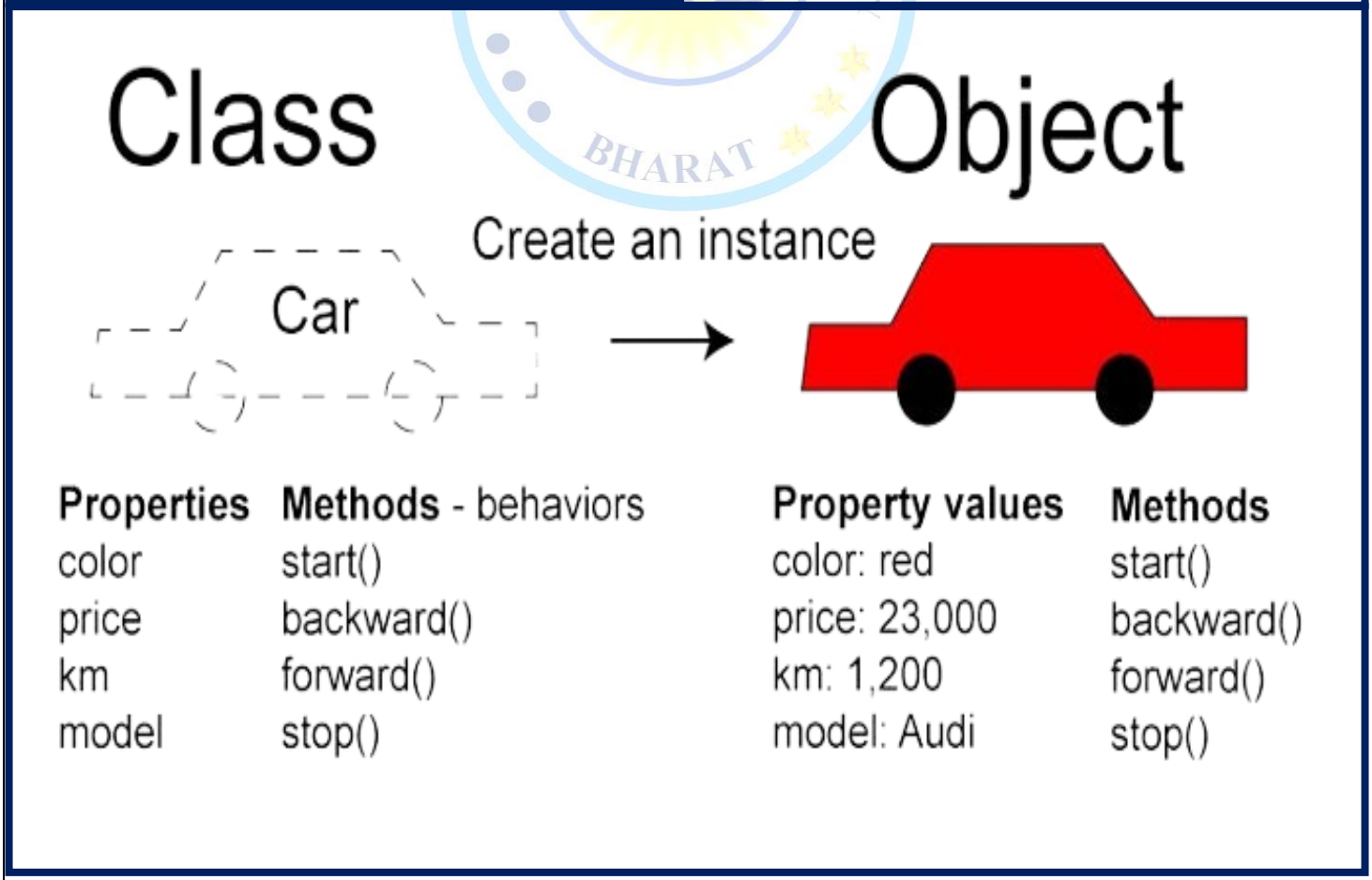


2. Class:

- Collection of **objects** is called class. It is a logical entity.
- To create objects, we required some model or plan or template or blue print, which is nothing but class.
- We can write a class to represent **properties(attributes)** and **actions (behaviour)** of object.
- **Properties** can be represented by **variable**
- **Action** can be represented by **methods.**
- Hence class contains both **variables** and **method.**
- a class is a template definition of the methods and variables in a particular kind of object. Thus, an object is a specific **instance of a class**.

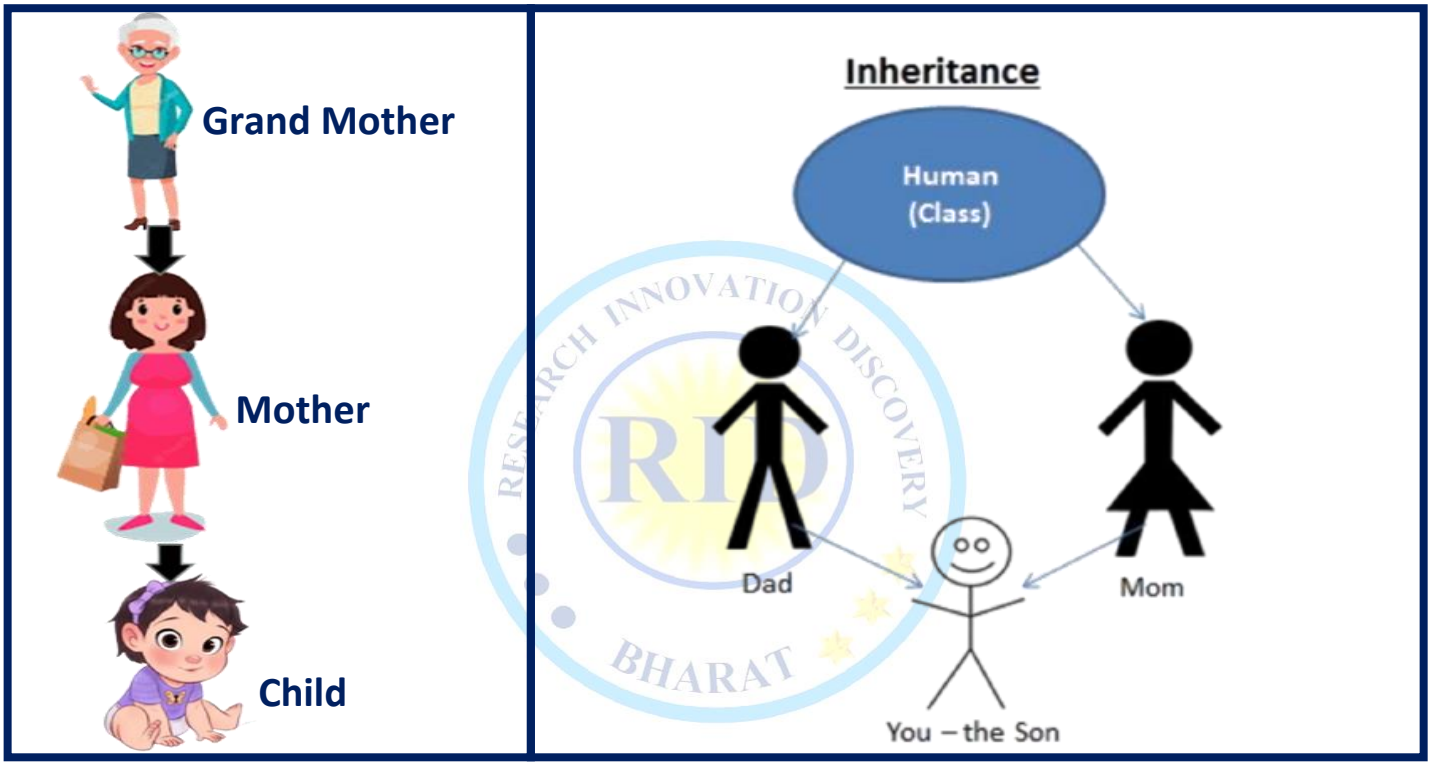


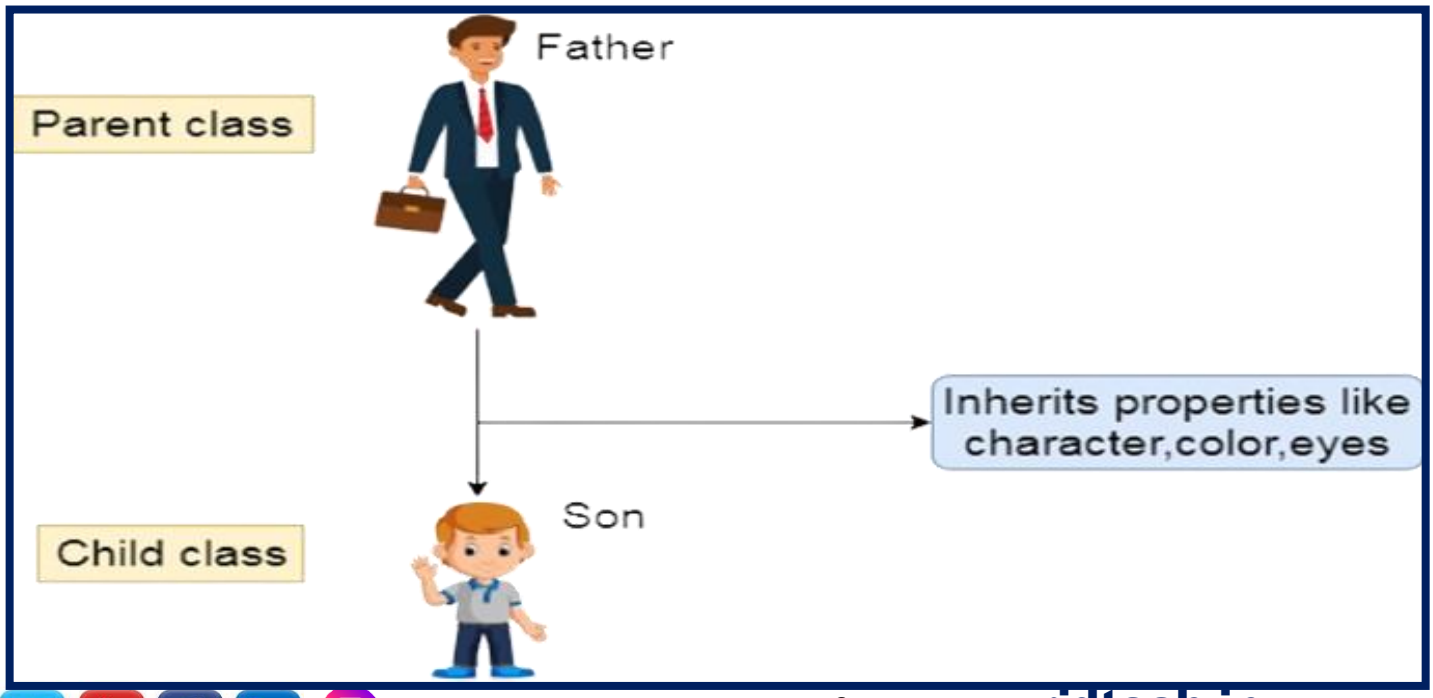




3. Inheritance:

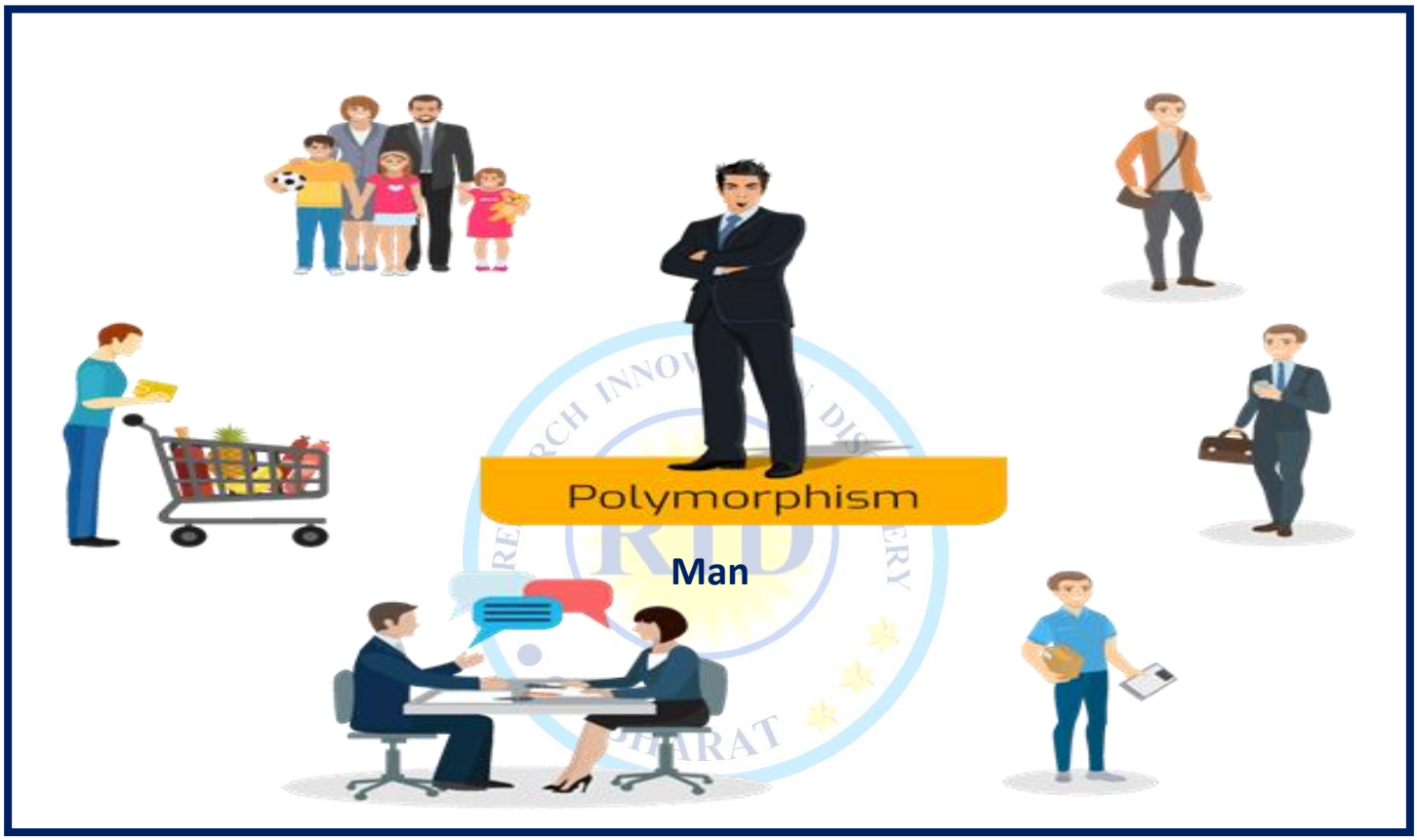
- When one object acquires all the **properties** and **behaviours** of **parent object** i.e. known as inheritance. It provides code reusability. It is used to achieve runtime **polymorphism.**
- **Sub class -** Subclass or Derived Class refers to a class that receives properties from another class.
- **Super class -** The term "Base Class" or "Super Class" refers to the class from which a subclass inherits its properties.
- **Reusability -** when we wish to create a new class, but an existing class already contains some of the code we need, we can generate our new class from the old class that is inheritance.

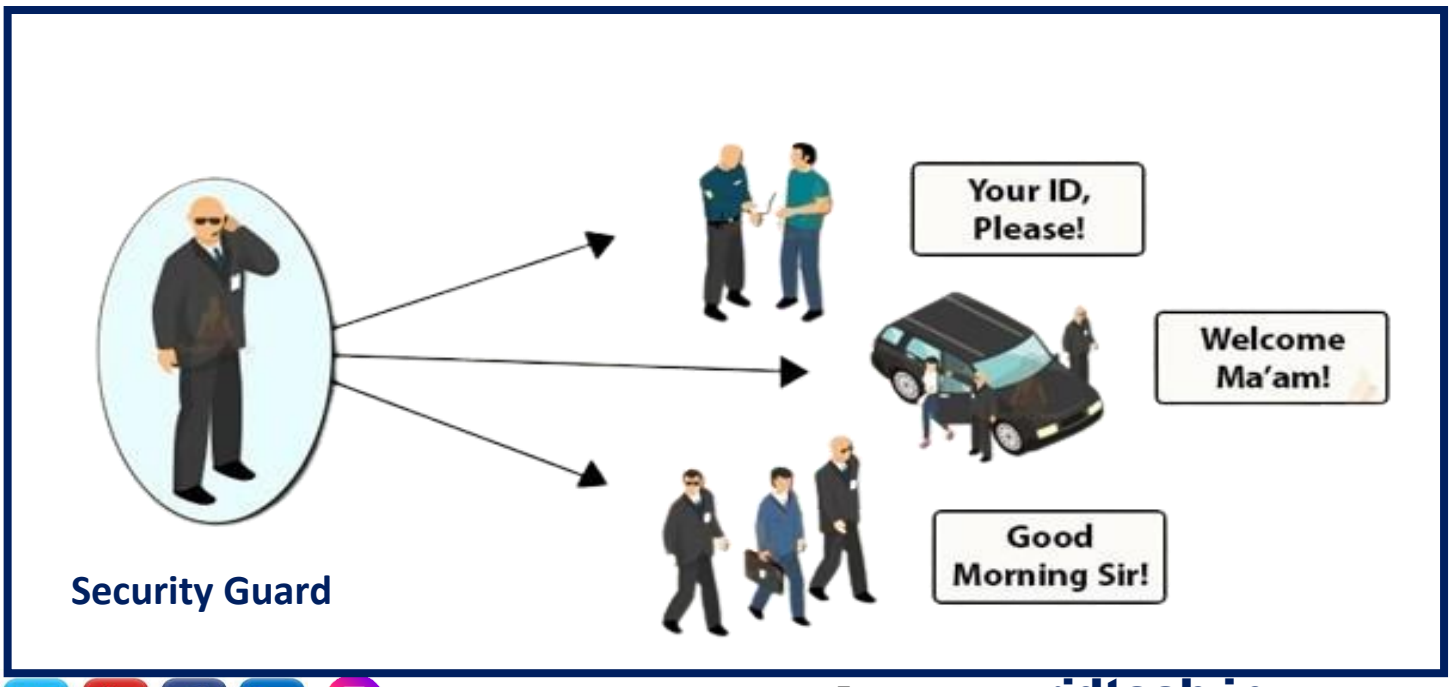




1. Polymorphism:

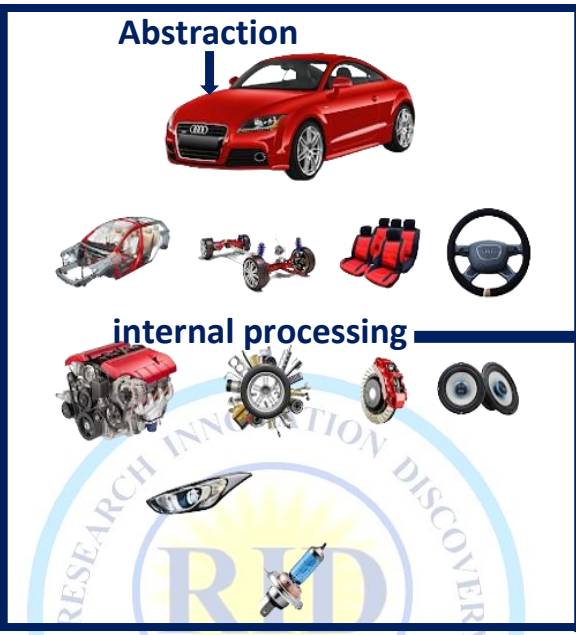
- When one task is performed by different ways i.e. known as polymorphism.
- Polymorphism means "many forms"
- Imagine a man named Sangam. Sangam has different roles depending on the context:
  - ✓ At work, Sangam is an Engineer.
  - ✓ At home, Sangam is a father.
  - ✓ In a social gathering, Sangam is a Friend.
  - ✓ In flight, Sangam is passenger.

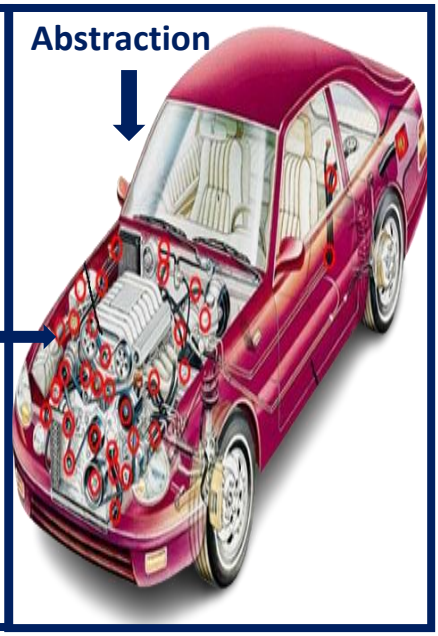


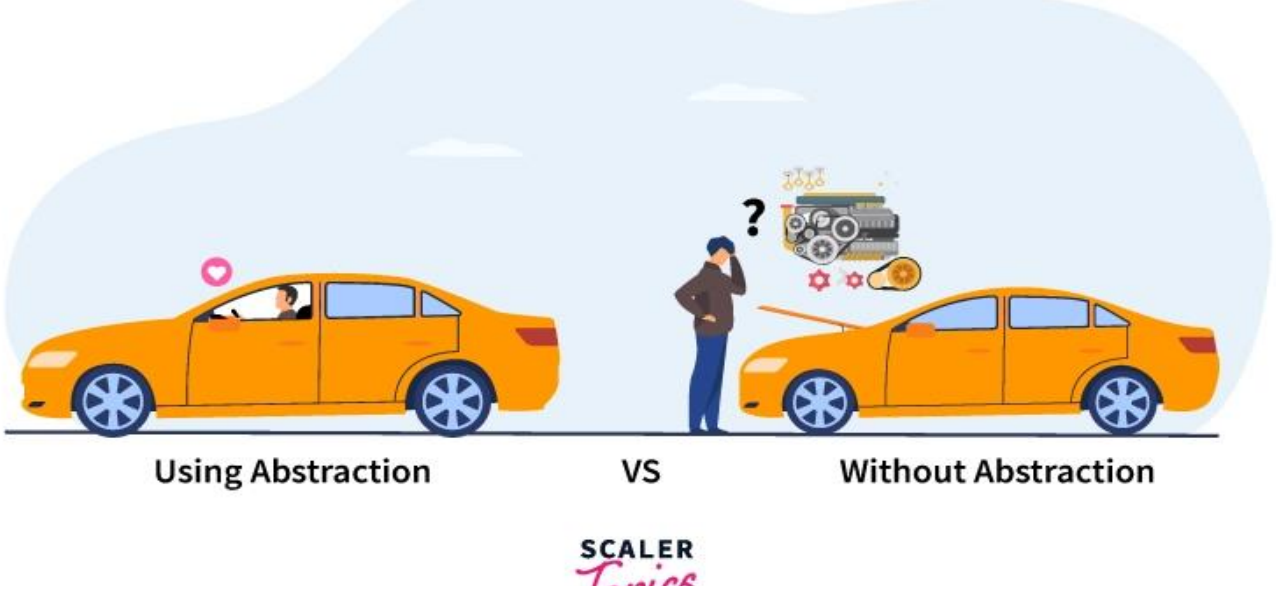


2. Abstraction:

- Hiding **internal** details and showing **functionality** is known as abstraction.
- Data abstraction is the process of exposing to the outside world only the information that is absolutely necessary while concealing implementation or background information. For example: phone call and Car. we don't know the internal processing.
- n C++, we use abstract class and interface to achieve abstraction.

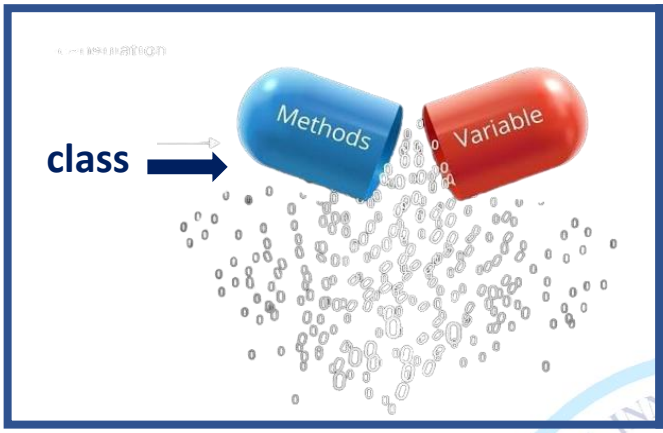


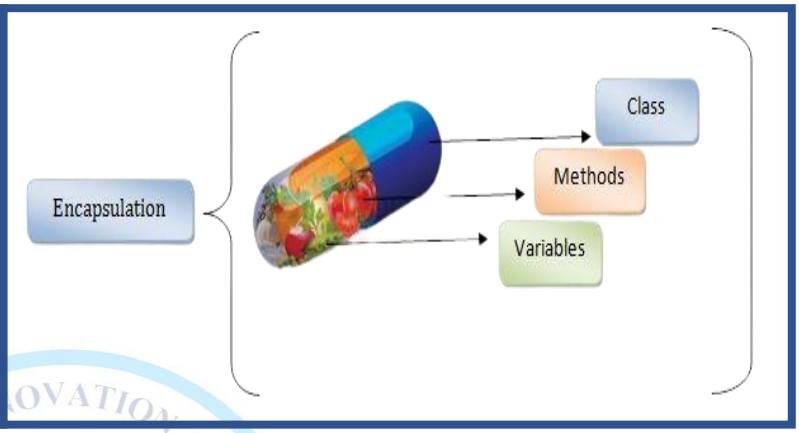


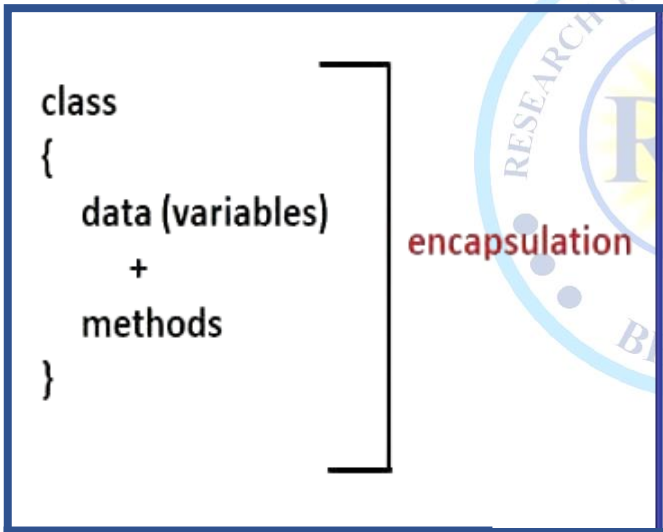


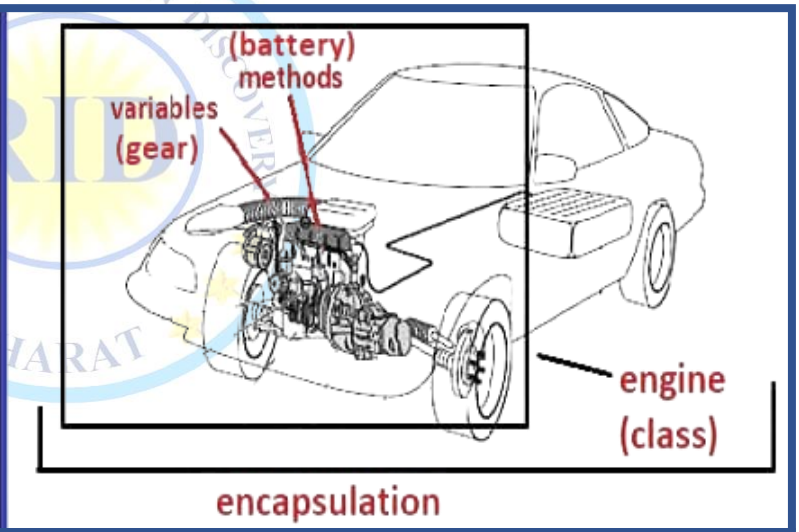
3. Encapsulation:

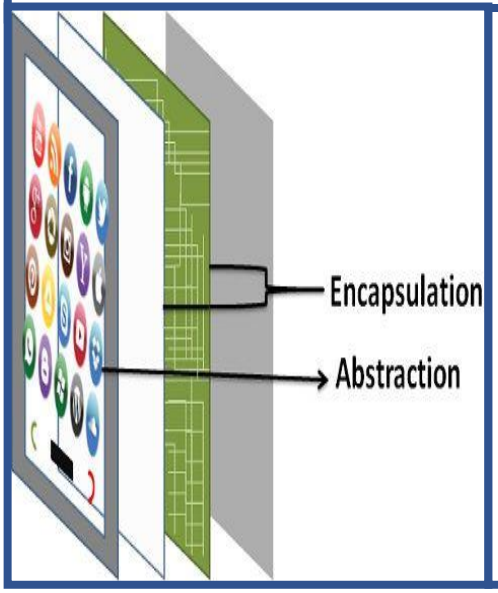
- **Binding** (or wrapping) **code** and **data** together into a single unit is known as encapsulation.
- encapsulation is described as the process of combining code (**methods or functions**) and data (**variables or attributes**) into a single unit.
- **Methods** or functions represent **executable** code, while data or variables represent **stored** information within a program.











Encapsulation:

For the mathematical equation, shown in the figure assume that complex functions are required - >But I the end we obtain result for it

Calculator shows the result of equation but hides the implementation (calculating the result) involved.

Abstraction:

The calculator shown in the figure has to be powered by a battery source. How the battery module works for the calculator is not necessary to know for the user who uses the calculator.

Using Battery module along with other modules we use calculator -> Thus using Abstraction encapsulation is performed

❖ Advantages and importance of OOPS

1. **Modularity and Code Organization:**
   - OOP promotes modularity by breaking down complex systems into smaller, manageable parts. This modularity makes it easier to develop, test, and maintain code.
2. **Reusability:**
   - OOP encourages the creation of reusable code components (classes and objects). These components can be used in different parts of the program or even in other projects.
3. **Encapsulation:**
   - Encapsulation ensures that the internal state and implementation details of an object are hidden from the outside world. This protects the integrity of data and allows for controlled access to object attributes.
4. **Inheritance:**
   - Inheritance allows for the creation of new classes by inheriting attributes and methods from existing classes. This promotes code reuse and the establishment of hierarchical relationships.
   - It enables the creation of specialized classes (subclasses) that inherit the properties of more general classes (superclasses).
5. **Polymorphism:**
   - Polymorphism enables objects of different classes to be treated as objects of a common superclass. This promotes flexibility and extensibility in code design.
6. **Abstraction:**
   - Abstraction simplifies complex systems by focusing on the essential features and ignoring unnecessary details. It helps in modeling real-world entities in a clear and understandable way.
7. **Improved Collaboration:**
   - OOP encourages the use of well-defined interfaces and contracts between objects and classes. This clear specification of interactions between components makes it easier for multiple developers to collaborate on large projects.
8. **Scalability and Maintainability:**
   - OOP principles make it easier to scale and maintain software systems as they grow in size and complexity.
9. **Real-World Modeling:**
   - OOP facilitates the modeling of real-world entities and their relationships, making it easier to translate real-world problems into code.
   - It helps in building software systems that align with real-world processes and concepts.
10. **Industry Standard:**
    - OOP is a widely adopted and industry-standard programming paradigm. Many programming languages, such as Python, Java, C++, and C#, are designed with OOP principles at their core.
    - Proficiency in OOP is a valuable skill for software developers, making it easier to collaborate on projects and find job opportunities in the software industry.

What is class?

- In python everything is an object. To create objects, we required some model or plan or template or blue print, which is nothing but class.
- We can write a class to represent properties(attributes) and actions (behaviour) of object.
- Properties can be represented by variable
- Action can be represented by methods.
- Hence class contains both variables and method

❖ How to define a class?

- We can define class by using class keyword.

Syntax:

class class_name:
'''documentation string'''

Variables: instance variables, static and local variables

Method: instance methods, static methods, static methods, class methods

- **Documentation string**: it is representing description of the class. Within class doc string is always optional. We can get doc string by using the flowing 2 ways

Example:

```
print(classname. __doc__)
Help(classname)
```

Example:

```
class skills:
 ''' this is skills class with details information'''
 print(skills.__doc__)
 help(student)
```

Example:

```
class  raj:
   """this it is raj class """
   print("hello ")
print(raj.__doc__)
#help(raj)
```

Output:

```
hello
this it is raj class
```

VARIABLE IN OOPS

➢ **Within the python class we can represent data by using variables**

There are 3 types of variables are allowed:

1. **Instance variables (object level variable)**
2. **Static variables (class level variables)**
3. **Local variables (method level variables)**

1. Instance Variable

- Belongs to each object.
- Defined using self inside methods.
- means with constructure first parameter defined instance variable

Syntax: self.variable_name = value

Example:

```
class Student:
    def __init__(self, name):
        self.name = name  # instance variable

s1 = Student("Sangam Kumar")
s2 = Student("Ankit")
print(s1.name)  # Sangam Kumar
print(s2.name)  # Ankit
```

2. Static Variable

- Belongs to the class (shared by all objects).
- Defined **inside class but outside methods**, or with ClassName.variable.

Syntax: ClassName.variable_name = value

Example:

```
class Student:
    school = "RID School"  # static variable

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

s1 = Student("Sangam")
s2 = Student("Ankit")
print(s1.school)  # RID School
print(s2.school)  # RID School
```

3. Local Variable

- Defined **inside a method**.
- Exists only during method execution.

Syntax:

```
def method_name(self):
    variable_name = value
```

Example:

```
class Student:
    def show(self):
        age = 15   # local variable
        print("Age is", age)

s1 = Student()
s1.show()  # Age is 15
```

Difference between instance variable static variable and local variable

Feature	Instance Variable	Static Variable	Local Variable
1. **Definition**	Belongs to an object	Belongs to a class	Declared inside a method
2. **Scope**	Accessible using self inside the class	Accessible using class name or object	Accessible only inside the method
2. Memory location	Created in object memory	Stored in class memory (shared)	Exists in method stack memory
3. When created	When an object is created	When the class is loaded	When method is called
4. Usage	To store object-specific data	To store common data for all objects	To perform temporary calculations
5. Access Syntax	self.var_name	ClassName.var_name or object.var_name	Just var_name inside method
6. Example	self.name = "Tanmay"	school = "RID"	marks = 90 inside a method

❖ Complete Example for all three variables

```
class Student:
    # Static variable (class level)
    school_name = "RID School"
    def __init__(self, name, age):
        # Instance variables (object level)
        self.name = name
        self.age = age
    def show_details(self):
        # Local variable (method level)
        grade = "10th"
        print("Name:", self.name)
        print("Age:", self.age)
        print("Grade:", grade)
        print("School:", Student.school_name)
# Creating objects
s1 = Student("Tanmay", 15)
s2 = Student("Ankit", 16)
# Calling method
s1.show_details()
print("------")
s2.show_details()
```

What this does:

- self.name and self.age are **instance variables**.
- school_name is a **static variable** shared by all students.
- grade is a **local variable** used only inside the method show_details().

METHOD IN OOPS

- **Within the python class we can represent operation by using methods. The following are various types of allowed methods**
  1. **Instance methods**
  2. **Class methods**
  3. **Static methods**

1. Instance Method

- **Definition:** Instance methods are used to access or modify object (instance) data. They take self as the first parameter.
- Works with instance variables of an object.
- First parameter is always self.

Syntax:
```
class ClassName:
    def instance_method(self):
        # access instance variables using self
```

Example:
```
class Student:
    def __init__(self, name, age):
        self.name = name     # instance variable
        self.age = age
    def display_info(self):  # instance method
        print("Name:", self.name)
        print("Age:", self.age)
# Creating object
s1 = Student("Sangam", 15)
s1.display_info()
```

Output: Name: Sangam
Age: 15

2. Class Method

- **Definition:** Class methods work with class-level data (static variables). They take cls as the first parameter and are defined using the @classmethod decorator.

Syntax:
```
class ClassName:
    @classmethod
    def class_method(cls):
        # access class variables using cls
```

Example:
```
class Student:
    school = "RID School"    # static (class) variable
    @classmethod
    def change_school(cls, new_name):  # class method
        cls.school = new_name
        print("School changed to:", cls.school)
Student.change_school("TWF School")
```

Output School changed to: TWF School

3. Static Method

- **Definition:** Static methods do not access instance or class data. They are used for utility tasks and are defined using the @staticmethod decorator.
- **Works with class variables (static variables).**
- **First parameter is always cls.**
- **Use @classmethod decorator.**

Syntax:

```
class ClassName:
    @staticmethod
    def static_method():
        # general utility code
```

Example:

```
class Student:
    @staticmethod
    def welcome_message():   # static method
        print("Welcome to the new academic year!")

Student.welcome_message()
```

Output: Welcome to the new academic year!

Decorator

Decorator (in Python):- A **decorator** is a special function that **adds extra features** to another function or method **without changing its code**.

Syntax:

```
@decorator_name
def my_function():
    # original code
```

- **Example:**

```
def greet_decorator(func):
    def wrapper():
        print("Hello!")
        func()
        print("Goodbye!")
    return wrapper

@greet_decorator
def say_name():
    print("My name is Tanmay")

say_name()
```

Output:

```
Hello!
My name is Tanmay
Goodbye!
```

- **In OOPs:**
  - @classmethod → works with class (cls)
  - @staticmethod → regular method with no self or cls

Example of Class, Instance, and Class Variables

```
class A:  # Creating a class
   x = 30  # Class variable (same for all objects)

   def __init__(self, a, b):  # Constructor with instance variables
      self.a = a  # Instance variable
      self.b = b  # Instance variable

   def dis(self):  # Instance method
      print("Instance Variables:", self.a, self.b)
      print("Class Variable (via self):", self.x)
      print("Class Variable (via class):", A.x)

# Creating object
ob = A(10, 20)

# Accessing class and instance details
print(type(ob))       # <class '__main__.A'>
print(ob.x)           # Access class variable using object
print(A.x)            # Access class variable using class name
print(ob.a, ob.b)     # Access instance variables using object

# Creating more objects
ob1 = A(20, 30)
ob2 = A(30, 40)

# Accessing instance variables of all objects
print(ob1.a, ob1.b)
print(ob2.a, ob2.b)

# Calling method to display all info
ob.dis()
```

Note:

- Class variable can be accessed by class name, self, or object
- Instance variables can only be accessed using object or self
- Class name cannot access instance variables directly

Quick Summary:

1. **Class Variable (x)** — shared by all objects
2. **Instance Variables (a, b)** — unique to each object
3. **Access:**
    - A.x, self.x, object.x
    - self.a, object.a
    - A.a

Example of Instance vs Class vs Static Methods

```
class A:
    x = 30  # Class variable

    def __init__(self, a, b):  # Constructor
        self.a = a          # Instance variable
        self.b = b

    def dis(self):  # Instance Method
        print("Instance variables:", self.a, self.b)
        print("Access class variable in instance method:", self.x)

    @classmethod
    def clsmethod(cls):  # Class Method
        print("Access class variable in class method:", cls.x)

    @staticmethod
    def stmethod():  # Static Method
        print("Access class variable in static method:", A.x)

# Create object
ob = A(10, 20)

# Call methods
ob.dis()       # Instance method
ob.clsmethod()  # Class method
ob.stmethod()   # Static method
```

Example of Class, Instance, & Class Variables and Instance, Class, Static Methods

Example:

```
class Example:
    # Class variable (shared by all objects)
    count = 0
    def __init__(self, name):
        # Instance variable (unique for each object)
        self.name = name
        Example.count += 1  # Increase count when a new object is created
    # Instance method (access instance and class variables)
    def show(self):
        print(f"Name: {self.name}")
        print(f"Total objects created: {Example.count}")

    # Class method (access class variable only)
```

```
    @classmethod
    def show_count(cls):
        print(f"Total objects (from class method): {cls.count}")

    # Static method (general utility, no access to instance or class variables)
    @staticmethod
    def greet():
        print("Hello! Welcome to the Example class.")

# Create objects
obj1 = Example("Sangam")
obj2 = Example("Bob")

# Call instance method
obj1.show()

# Call class method (can be called using class or object)
Example.show_count()
obj2.show_count()

# Call static method (can be called using class or object)
Example.greet()
obj1.greet()
```

Output:

```
Name: Sangam
Total objects created: 2
Total objects (from class method): 2
Total objects (from class method): 2
Hello! Welcome to the Example class.
Hello! Welcome to the Example class.
```

This example shows:

- How **instance variables** (self.name) work per object.
- How **class variable** (count) is shared by all objects.
- How **instance method** uses both instance and class variables.
- How **class method** accesses class variable only.
- How **static method** works independently of instance/class variables.

What is Object

- An **object** is a **real-world instance** of a class.
  It is created from a class and can be used to **access class variables and methods**.
- You can create **multiple objects** from a single class.
- Physical existence of a class is nothing but object. We can create any number of objects for a class

1. An object is an instance of a class.
2. It gives physical existence to a class.
3. Multiple objects can be created from one class.
4. Each object has its own set of instance variables.
5. Objects are created using the class name (constructor).
6. Reference variables are used to access objects.
7. Objects can access class and instance variables, and methods.
8. Objects use the dot (.) operator to access members.
9. Data inside objects is stored separately.
10. Objects support encapsulation in OOP.

Syntax: object_name = ClassName()

- **Object** = Instance of a class
- **Reference variable** = Name used to access the object

Example:

```
class Person:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print(f"Hello, my name is {self.name}")
p1 = Person("Alice")  # Object created
p2 = Person("Bob")    # Another object
p1.greet()
p2.greet()
```

❖ What is reference variable?

- The variable which can be used to refer object is called reference variable.
- By using reference variable, we can access properties and methods of object.

Example-1:

```
class Student:
    def __init__(self, name):
        self.name = name
    def show(self):
        print("Student name is:", self.name)
# 's' is a reference variable pointing to the object of Student
s = Student("Sangam ")  # ☜ Object created and 's' refers to it

# Using reference variable to access method and variable
s.show()        # Access method
print(s.name)   # Access instance variable
```

Output:

```
Student name is: Sangam
sangam
```

Example 2: Simple Object Creation

```
class Student:
    def study(self):
        print("Student is studying.")

# Creating object
s1 = Student()  # s1 is the object
s1.study()      # Calling method using object
```

Example 3: Object with Instance Variables

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

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

# Creating multiple objects
car1 = Car("Toyota", "Red")
car2 = Car("Honda", "Blue")

car1.show()
car2.show()
```

Example 4: Accessing Variables and Methods

```
class Dog:
    species = "Canine"  # Class variable

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

    def bark(self):
        print(f"{self.name} says Woof!")

# Creating object
d = Dog("Buddy")
print(d.name)     # Access instance variable
print(d.species)  # Access class variable
d.bark()          # Access method
```

Problem: Write a python program to create a student class and creates an object to it call the method skills() to display student details

Program:

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

```
    def skills(self):
        print("Hello, my name is:", self.name)
        print("My Roll No is:", self.rollno)
        print("My Marks are:", self.marks)

# Creating object and calling method using reference variable
obj1 = Student("Sangam", 39, 100)
obj1.skills()
```

Output:

```
Hello, my name is: Sangam
My Roll No is: 39
My Marks are: 100
```

Problem: Write a Python program to create a Car class and create an object to it. Call the method details() to display car information.

Example:

```
class Car:
    def __init__(self, brand, model, price):
        self.brand = brand
        self.model = model
        self.price = price

    def details(self):
        print("Car Brand:", self.brand)
        print("Car Model:", self.model)
        print("Car Price:", self.price)

# Creating an object and calling the method
car1 = Car("Toyota", "Fortuner", 3500000)
car1.details()
```

Output:

```
Car Brand: Toyota
Car Model: Fortuner
Car Price: 3500000
```

❖ Self-Variable

What is self in Python?

- self is a **default variable** used inside class methods and constructor.
- It always **refers to the current object** of the class.
- It is similar to this keyword in Java.
- With self, you can **access instance variables and methods**.

🔸 **Important Notes:**

1. self must be the **first parameter** of the constructor (__init__)
   👉 Example: def __init__(self):
2. self must be the **first parameter** in instance methods
   👉 Example: def display(self):

Example 1: Simple use of self in constructor and method

```
class Person:
    def __init__(self, name):
        self.name = name  # using self to store value in instance variable

    def greet(self):
        print("Hello, my name is", self.name)

# Creating object
p = Person("Tanmay")
p.greet()
```

Output: Hello, my name is Tanmay

Example 2: Using self to access multiple instance variables

```
class Laptop:
    def __init__(self, brand, price):
        self.brand = brand
        self.price = price

    def show_info(self):
        print("Brand:", self.brand)
        print("Price:", self.price)

l1 = Laptop("HP", 55000)
l1.show_info()
```

Output:

```
Brand: HP
Price: 55000
```

Example 3: Using self to call another method from a method

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

    def display(self):
        print("Book Title:", self.title)
        self.thanks()  # calling another method using self

    def thanks(self):
        print("Thanks for checking out the book!")

b = Book("Python Basics")
b.display()
```

Output:

```
Book Title: Python Basics
Thanks for checking out the book!
```

Constructor

- A **constructor** in Python is a special method used to initialize the instance variables of a class. It is automatically called **when an object is created**. The constructor method in Python is always named __init__().

➢ **Special Method**:
A constructor is a special method used to initialize objects in Python.

➢ **Method Name**:
The constructor method must be named __init__(self).

➢ **Automatic Execution**:
The constructor is executed automatically when a new object of the class is created.

➢ **Purpose**:
Its primary purpose is to declare and initialize instance variables for the object.

➢ **One-Time Execution Per Object**:
For each object, the constructor is called only once—during its creation.

➢ **Arguments**:
A constructor must take at least one argument, typically self, which refers to the current instance.

➢ **Optional in Nature**:
Defining a constructor is optional. If no constructor is defined, Python automatically provides a **default constructor**.

Constructor Syntax

```
class ClassName:
    def __init__(self):
        # Initialization code
        self.variable = value
```

❖ Types of Constructors in Python

1. **Default Constructor**
   - Constructor with no parameters (except self)
   - Automatically provided by Python if no __init__ is defined
2. **Parameterized Constructor**
   - Takes arguments to initialize variables dynamically

Example-1: Default Constructor

```
class MyClass:
    def __init__(self):
        # Default constructor
        self.message = "Hello sangam"

    def display(self):
        print(self.message)

obj = MyClass()
obj.display()
```

Example-2: Parameterized Constructor

```
class MyClass:
    def __init__(self, name, age):
        # Parameterized constructor
        self.name = name
        self.age = age

    def display(self):
        print("Name:", self.name)
        print("Age:", self.age)

obj = MyClass("Sangam", 20)
obj.display()
```

1. Simple Constructor (No Parameters)

```
class Student:
    def __init__(self):
        print("Constructor called")
        self.name = "Sangam Kumar"

    def show(self):
        print("Name:", self.name)

# Creating object
s = Student()
s.show()
```

Output:

```
Constructor called
Name: Sangam kumar
```

2. Constructor with Parameters

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

    def show(self):
        print("Name:", self.name)
        print("Age:", self.age)

# Creating object with parameters
s1 = Student("Sangam", 15)
s1.show()
```

Output:

```
Name: Sangam
Age: 15
```

3. Multiple Objects Using the Same Constructor

```
class Car:
    def __init__(self, model):
        self.model = model

    def display(self):
        print("Car Model:", self.model)

car1 = Car("Toyota")
car2 = Car("Honda")

car1.display()
car2.display()
```

Output: Car Model: Toyota

```
Car Model: Honda
```

Program: program to demonstrate constructor execute will execute only once per object.

```
class demo:
    def _init_(self):
        print("constructor execution")
    def method1(self):
        print("method execution")
obj1=demo()
obj2=demo()
obj3=demo()
obj1.method1()
```

Output: constructor execution
constructor execution
constructor execution
method execution

Problem: Write a program display the student details:

Program:

```
class student:
    This is student class with required data #docstring
    def _init_(self,a,b,c):
        self.name=a
        self.rollno=b
        self.marks=c
    def result(self):
        print("Student Name:{}\n  Rollno:{}\n  Marks:{}".format(self.name,self.rollno,self.marks))
s1=student("Sangam Kumar", 39,100)
s1.result()
s2=student("Sataym Kumar", 103,99)
s2.result()
s3=student("Raushani Kumari", 53,98)
s3.result()
```

Output:

```
Student Name:Sangam Kumar
  Rollno:39
  Marks:100
Student Name:Sataym Kumar
  Rollno:103
  Marks:99
Student Name:Raushani Kumari
  Rollno:53
  Marks:98
```

❖ Difference between methods and constructors:

Methods:

1. Name of method can be any name
2. Method will be executed if we call that method
3. per object, method can be called any number of times
4. inside method we can write business logic

constructor:

1. Constructor name should be always _init_
2. constructor will be executed automatically at the time of object creation
3. per object, constructor will be executed only once
4. inside constructor we have to declare and initialize variables

1). instance variables:

- if the value of a variable i varied from object to object, then such type of variables is called instance variables.

➢ for every object a separate copy of instance variables will be created.

➢ where we can declare instance variables:

1. inside constructor by using self-variable
2. inside instance method by using self-variable
3. outside of the class by using object reference variable

1. inside constructor by using self-variable:

- we can declare instance variables inside a constructor but using self-keyword. once we create object, automatically these variables will be added to the object.

Example:

```
class employee:
    def _init_(self):
        self.eno=103
        self.ename='sujeet Kumar'
        self.esal=100000
r=employee()
print(r._dict_)
```

output:

```
{'eno': 103, 'ename': 'sujeet Kumar', 'esal': 100000}
```

2). inside instance method by using variable:

- we can also declare instance variables inside instance method by using self-variable. if any instance variable declared inside instance method, that instance variable will be added once we call that method.

Example:

```
class test:
    def _init_(self):
        self.a=30
        self.b=20
    def m1(self):
        self.c=40 # here we are declaring instance variables inside instance method by using
self-variable for c.
obj=test()
obj.m1()
print(obj._dict_)
```

output:

```
{'a': 30, 'b': 20, 'c': 40}
```

3). outside of the class but using object reference variable:

- we can also add instance variables outside of a class to particular object.

Example:

```
class test:
    def _init_(self):
        self.a=20
        self.b=30
```

```
    def m1(self):
        self.c=40
obj=test()
obj.m1()
obj.d=50 # here we are adding instance variables outside of a class to particular object d
print(obj._dict_)
```

output:

```
{'a': 20, 'b': 30, 'c': 40, 'd': 50}
```

❖ How to access instance variables:

- we can access instance variables with in the class by using self-variable and outside of the class by using object reference.

Example:

```
class test:
    def _init_(self):
        self.a=20
        self.b=30
    def display(self):
        print(self.a)
        print(self.b)
r=test()
r.display()
print(r.a,r.b)
```

output:

```
20
30
20 30
```

❖ How to delete instance variable from the object:

1). within a class we can delete instance variable as follows:

```
del self.varaibleName
```

2). From outside of class, we can delete instance variable as follows:

```
del objectreference.variableName
```

Example:

```
class test:
    def _init_(self):
        self.a=10
        self.b=20
        self.c=30
        self.d=40
    def method1(self):
        del self.d #within a class we can delete instance variable
t=test()
print(t._dict_)
t.method1()
print(t._dict_)
del t.c #outside of class we can delete instance variable
print(t._dict_)
```

output:

```
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30}
{'a': 10, 'b': 20}
```

Note: the instance variables which are deleted from one object, will not be deleted from other object.

Example:

```
class test:
   def _init_(self):
      self.a=10
      self.b=20
      self.c=30
      self.d=40
obj1=test()
obj2=test()
del obj1.a
print(obj1._dict_)
print(obj2._dict_)
```

output:

```
{'b': 20, 'c': 30, 'd': 40}
{'a': 10, 'b': 20, 'c': 30, 'd': 40}
```

- if we change the value of instance variables of one object then those changes won't be reflected to the remaining objects because for every object we are separate copy of instance variables are available.

Example:

```
class test:
   def _init_(self):
      self.a=10
      self.b=20
      self.c=30
r1=test()
r1.a=393
r1.b=333
r2=test()
print('r1:',r1.a, r1.b)
print('r2:',r2.a, r2.b)
```

output: r1: 393 333

r2: 10 20

2) Static Variables:

- if the value of a variable is not varied from object to object, such type of variable we have to declare with in the class directly but outside of methods. such types of variables are called static variables.
- for total class only one copy of static variable will be created and shared by all objects of that class.
- we can access static variables either by class name or by object reference. But recommended to use class name.

❖ instance variable vs static variable:

Note: in the case of instance variable for every object a separate copy will be created, but in the case of static variables for total class only one copy will be created and shared by every object of that class.

Example:

```
class demo:
   a=10
   def _init_(self):
      self.b=20
r1=test()
r2=test()
print('r1:',r1.a, r1.b)
print('r2:',r2.a, r2.b)
demo.a=393
r1.b=33
print('r1:',r1.a, r1.b)
print('r2:',r2.a, r2.b)
```

output:

```
r1: 10 20
r2: 10 20
r1: 10 33
r2: 10 20
```

- where we can declare static variables:
- in various places to declare static variables:
  1). in general, we can declare within the class directly but from outside of any method
  2). inside constructor but using class name
  3). inside instance method by using class name
  4). inside class method by using either class name or class variable
  5). inside static method by using class name.

Example:

```
class Demo:
   a=10
   def _init_(self):
      Demo.b=20
   def m1(self):
      Demo.c=30
   @classmethod
   def m2(skills):
      skills.d=40
      Demo.d1=300
   @staticmethod
   def m3():
      Demo.e=50
print(Demo._dict_)
r=Demo()
print(r.b)
print(Demo._dict_)
Demo.m2()
```

```
print(Demo._dict_)
Demo.m3()
print(Demo._dict_)
demo.f=60
print(Demo._dict_)
_repr_
```

❖ How to access static Variables:

1. inside constructor: by using either self or classname
2. inside instance method: by using either self or classname
3. inside static method: by using classname
4. inside class method: by using either class variable or classname
5. from outside of class: by using either reference or classname

Example:

```
class Demo:
   a=10
   def _init_(self):
      print(self.a)
      print(Demo.a)
   def m1(self):
      print(self.a)
      print(Demo.a)
   @classmethod
   def m2(cls):
      print(cls.a)
      print(Demo.a)
   @staticmethod
   def m3():
      print(Demo.a)
t=Demo()
print(Demo.a)
print(t.a)
t.m1()
t.m2()
t.m3()
```

output:

```
10
10
10
10
10
10
10
10
10
```

❖ where we can modify the value of static variable:

- anywhere either with in the class or outside of class we can modify by using classname. But inside class method, by using class variable

Example:

```
class test:
   a=666
   @classmethod
   def m1(cls):
      cls.a=333
   @staticmethod
   def m2():
      test.a=121
print(test.a)
test.m1()
print(test.a)
test.m2()
print(test.a)
```

output: 666
333
121

- if we change the value of static variable by using either self or object reference variable:
- if we change the value of static variable by using either self or object reference variable, then the value of static variable won't be changed, just a new instance variable with that name will be added to that particular object.

Example:

```
class test:
   a=10
   def m1(self):
      self.a=333
t1=test()
t1.m1()
print(test.a)
print(t1.a)
```

output: 10, 333

Example:

```
class test:
   a=20
   def _init_(self):
      self.b=30
r1=test()
r2=test()
print('r1:',r1.a, r1.b)
print('r2:',r2.a, r2.b)
r1.a=333
r2.b=666
print('r1:',r1.a, r1.b)
print('r2:',r2.a, r2.b)
```

output: r1: 20 30
r2: 20 30
r1: 333 30
r2: 20 666

Example:

```
class test:
   a=20
   def _init_(self):
      self.b=30
r1=test()
r2=test()
test.a=333
r1.b=666
print('r1:',r1.a, r1.b)
print('r2:',r2.a, r2.b)
```

output: r1: 333 666
r2: 333 30

Example:

```
class test:
   a=25
   def _init_(self):
      self.b=50
   def m1(self):
      self.a=333
      self.b=999
r1=test()
r2=test()
r1.m1()
print('r1:',r1.a, r1.b)
print('r2:',r2.a, r2.b)
```

output: r1: 333 999
r2: 25 50

Example:

```
class test:
   a=25
   def _init_(self):
      self.b=50
   @classmethod
   def m1(skills):
      skills.a=333
      skills.b=999
r1=test()
r2=test()
r1.m1()
print('r1:',r1.a, r1.b)
print('r2:',r2.a, r2.b)
print(test.a, test.b)
```

output: r1: 333 50
r2: 333 50
333 999

❖ How to delete static variables of a class:

1). we can delete static variables from anywhere by using the following syntax:

del classname.variablename

2). But inside classmethod we can also use class variable

del class.variablename

Example:

```
class demo:
   a=30
   @classmethod
   def m1(skills):
      del skills.a
demo.m1()
print(demo._dict_)
```

3). Local Variables:

- sometimes to meet temporary requirement of programmer, we can declare variable inside a method directly, such type of variables are called local variable or temporary variables.
- local variables will be created at the time of method execution and destroyed once method completes.
- local variables of a method cannot be accessed from outside of method.

Example:

```
class test:
   def m1(self):
      a=666
      print(a)
   def m2(self):
      b=333
      print(b)
t=test()
t.m1()
t.m2()
```

output:

```
666
333
```

Example:

```
class test:
   def m1(self):
      a=333
      print(a)
   def m2(self):
      b=666
      print(a) #NameError: name 'a' is not defined
      print(b)
t=test()
t.m1()
t.m2()
```

output:

```
333
666
```

❖ setter and getter methods:

What are they?

- **Setter methods** are used to **set (update)** values of instance variables.
- **Getter methods** are used to **get (access)** values of instance variables.
- This is a way to follow **encapsulation** in object-oriented programming.

Setter Method

- Used to set/update the value of an instance variable.
- Also known as a **mutator** method.

Syntax:

```
def setVariable(self, value):
    self.variable = value
```

Example:

```
class Student:
    def setName(self, name):
        self.name = name
```

Getter Method

- Used to access the value of an instance variable.
- Also known as an **accessor** method.

Syntax:

```
def getVariable(self):
    return self.variable
```

Example:

```
class Student:
    def setName(self, name):
        self.name = name

    def getName(self):
        return self.name
```

Complete Example

```
class Student:
    def setName(self, name):
        self.name = name

    def getName(self):
        return self.name

# Create object
s = Student()

# Set name using setter
s.setName("Tanmay")

# Get name using getter
print("Student Name:", s.getName())
```

Output: Student Name: Tanmay

? Are setName and getName mandatory names?

- **No, they are not mandatory.**
  In Python, **you can name the methods anything you like**, as long as you follow Python's naming rules.
- setName and getName are just **common naming conventions** to make your code more understandable.

Example: You can rename them to anything, like:

```
class Student:
    def assign_name(self, name):
        self.name = name

    def fetch_name(self):
        return self.name

# Create object
s = Student()

s.assign_name("Tanmay")
print("Student Name:", s.fetch_name())
```

Example:

```
class student:
    def setName(self,name):
        self.name=name
    def getName(self):
        return self.name
    def setMarks(self, marks):
        self.marks=marks
    def getMarks(self):
        return self.marks

n=int(input("Enter the Number of students: "))
for i in range(n):
    s=student()
    name=input("Enter The Name: ")
    s.setName(name)
    marks=int(input("Enter the marks: "))
    s.setMarks(marks)

    print("Hi", s.getName())
    print('Your Marks are:', s.getMarks())
    print()
```

output:

```
Enter the Number of students: 3
Enter The Name: Sushil Kumar
Enter the marks: 83
Hi Sushil Kumar
Your Marks are: 83
```

2) class methods:

- inside method implementation if we are using only class variable (static variables), then such type of methods we should declare as class method.
- we can declare class method explicitly by using @classmethod decorator.
- for class method we should provide cls variable at the time of declaration
- we can call classmethod by using classname or object reference variable.

Example:

```
class animal:
    a=4
    @classmethod
    def walk(self, name):
        print('{} walks with {} a...'.format(name,self.a))
animal.walk('Dog')
animal.walk('Cat')
```

output:

```
Dog walks with 4 a...
Cat walks with 4 a...
```

Program to track the Number of objects creates for a class:

```
class raj:
    count=0
    def _init_(self):
        raj.count=raj.count+1
    @classmethod
    def noOfObjects(self):
        print('The number of objects created for raj class:',self.count)
obj1=raj()
obj2=raj()
raj.noOfObjects()
r3=raj()
r4=raj()
r5=raj()
r6=raj()
raj.noOfObjects()
```

output:

```
The number of objects created for raj class: 2
The number of objects created for raj class: 6
```

3). static Methods:

- in general, these methods are general utility methods.
- inside these methods we won't use any instance or class variables.
- here we won't provide self or cls arguments at the time of declaration.
- we can declare static method explicitly by using @staticmethod decorator
- we can access static methods by using classname or object refernce.

Example:

```
class t3skills:
    @staticmethod
    def add(a,b):
        print('The sum:', a+b)
    @staticmethod
    def product(a,b):
        print('The product:',a*b)

    @staticmethod
    def average(a,b):
        print('The average:',(a+b)/2)
t3skills.add(25,50)
t3skills.product(3,6)
t3skills.add(20,30)
```

output:

```
The sum: 75
The product: 18
The sum: 50
```

Note: in general, we can use only instance static methods. inside static method we can access class level variables by using class name.

- class methods are most rarely used methods in python.

❖ passing members of one class to another class:

- we can access members of one class inside another class

Example:

```
class Employee:
   def _init_(self,eno,ename, esal):
      self.eno=eno
      self.ename=ename
      self.esal=esal
   def display(self):
      print("Employee Number:",self.eno)
      print("Employee Name:",self.ename)
      print("Employee salary:", self.esal)
class test:
   def modify(emp):
      emp.esal=emp.esal+20000
      emp.display()
e=Employee(100,'raj',30000)
test.modify(e)
```

output:

```
Employee Number: 100
Employee Name: raj
Employee salary: 50000
```

❖ inner classes:

- sometimes we can declare a class inside another class, such type of classes are called inner classes.
- without existing one type of object if there is no chance to existing another type of object then we should go for inner classes.

Example:

```
class car:
   .....
   class Engine:
   ......
```

Example:

- without existing university object there is no chance of existing department object:

```
class university:
   .....................
   class department:
   .....................
```

Example: without existing human there is no chance of existing head. hence head should be part of **human.**

```
class human:
   class head:
```

Note: without existing outer class object there is no chance of existing inner class object.

➢ hence inner class object is always associated with outer class object.

Program:

```
class outer:
```

```
    def _init_(self):
        print("outer class object creation")
    class inner:
        def _init_(self):
            print("inner class object creation")
        def m1(self):
            print("inner class method")
o=outer()
i=o.inner()
i.m1()
```

output:

```
outer class object creation
inner class object creation
inner class method
```

Note: the following are various possible syntaxes for calling inner class method

```
1) o=outer()
i=o.inner()
i.m1()
2) i=outer().inner()
i.m1()
3) outer().inner().m1()
```

Program:

```
class person:
    def _init_(self):
        self.name='t3 skills center'
        self.db=self.Dob()
    def display(self):
        print('Name:',self.name)
    class Dob:
        def _init_(self):
            self.dd=30
            self.mm=9
            self.yy=2023
        def display(self):
            print("Foundation Day={}-{}-{}".format(self.dd, self.mm, self.yy))
p=person()
p.display()
x=p.db
x.display()
```

output:

Name: t3 skills center
Foundation Day=30-9-2023

Garbage collection:

What is Garbage Collection in Python?

- **In old languages like C++:**
  - The programmer is responsible for both **creating** and **destroying** objects.
  - While programmers carefully create objects, they often **forget to destroy unused objects**.
  - This can lead to **memory getting full** with useless objects, causing the program to crash with an **"Out of Memory"** error.
- **In Python:**
  - Python has a helper called the **Garbage Collector**.
  - It runs in the **background** and automatically destroys **useless (unreferenced) objects**.
  - This helps avoid memory problems, so Python programs are **less likely to crash** due to memory issues.
- **Main Objective of Garbage Collector**
  - To **destroy useless objects** and **free up memory**.
  - Any object **without a reference** (i.e., not stored in any variable) is considered **eligible for garbage collection**.

How to Enable or Disable Garbage Collector in Python

- Python provides a built-in module called gc to control garbage collection manually.

Useful Functions:

Function	Description
gc.isenabled()	Checks if garbage collection is enabled (True or False)
gc.disable()	Disables garbage collection
gc.enable()	Enables garbage collection

Example Code

```
import gc
print(gc.isenabled())  # Check if GC is enabled (True by default)
gc.disable()           # Disable garbage collector
print(gc.isenabled())  # Should print False
gc.enable()            # Enable it again
print(gc.isenabled())  # Should print True
```

Output: True
False
True

What's Happening Here:

1. We created an object of class Demo.
2. We deleted the reference using del obj.
3. Since the object has no reference, it's **eligible for garbage collection**.
4. We called gc.collect() to **force garbage collection**.
5. The __del__() method is called, printing:

Example: Garbage Collection of Unused Object

```
import gc
class Demo:
   def __del__(self):
      print("Object destroyed")

# Create an object
obj = Demo()

# Delete the reference
del obj
# Force garbage collection
gc.collect()
```

Example:- Object destroyed

Destructions

What is it?

- **Destructor** is a **special method** in Python named __del__(self).
- It is called **just before an object is destroyed** by the **Garbage Collector**.
- The main purpose of the destructor is to **perform clean-up tasks** (like closing files, database connections, etc.).
  - **Important Note:**
    The destructor **does not destroy** the object itself.
    It only **prepares** the object for destruction by releasing resources.

Syntax of Destructor

```
class ClassName:
    def __del__(self):
        # clean-up code here
        print("Destructor called")
```

Simple Example

```
class MyClass:
    def __init__(self):
        print("Object Created")
    def __del__(self):
        print("Destructor called - Cleaning up")
# Create an object
obj = MyClass()
# Delete the reference
del obj
# Output:
# Object Created
# Destructor called - Cleaning up
```

What's Happening:

1. __init__() runs when the object is created.
2. del obj removes the reference.
3. Python's Garbage Collector sees that the object is no longer needed.
4. It calls __del__() to clean up before destroying the object.

Example:

```
import time
class test:
    def __init__(self):
        print("object initialization...")
    def __del__(self):
        print("Fulfilling Last wish and performing clean up activities..")
r1=test()
r1=None
time.sleep(5)
print("End of application")
```

output:

```
object initialization...
Fulfilling Last wish and performing clean up activities..
End of application
```

Note: if the object does not contain any reference variable, then only it is eligible to GC. i.e., if the reference count is zero then only object eligible for GC.

Example:

```
import time
class demo:
   def __init__(self):
      print("Constructor execution...")
   def __del__(self):
         print("Destructor Execution...")
r1=demo()
r2=r1
del r1
time.sleep(6)
print("object not yet destroyed after deleting r1")
del r2
time.sleep(5)
print("object not yet destroyed after deleting r2")
print("i am trying to delete last reference variable...")
del r3 #NameError: name 'r3' is not defined
```

output:

```
Constructor execution...
object not yet destroyed after deleting r1
Destructor Execution...
object not yet destroyed after deleting r2
i am trying to delete last reference variable...
```

Example:

```
import time
class test:
   def __init__(self):
      print("Constructor Execution...")
   def __del__(self):
      print("Destructor Execution")
l=[test(),test(),test()]
del l
time.sleep(6)
print("End of application")
```

output:

```
Constructor Execution...
Constructor Execution...
Constructor Execution...
Destructor Execution
Destructor Execution
Destructor Execution
End of application
```

❖ How to find the number of references of an object:

- sys module contains getrefcount() function for this purpose.
- sys.getrefcount(objectreference)

Example:

```
import sys
class test:
   pass
t1=test()
t2=t1
t3=t1
t4=t1
t5=t1
r6=t1
print("Count=",sys.getrefcount(t1))
```

output:

```
Count= 7
```

Has-A Relationship
Is-A Relationship
Is-A vs HAS-A Relation

- using members o one class inside another class
- we can use members of one class inside another class by using the following ways
  1). by composition (Has-A Relationship)
  2). By inheritance(Is-A Relationship)

1). By composition (Has-A Relationship):

- By using class name or by creating object we can access members of one class inside another class is nothing but composition (Has-A Relationship)
- the main advantage of Has-A Relationship is code Reusability.

Example:

```
class Engine:
   a=20
   def __init__(self):
      self.b=30
   def m1(self):
      print("Engine Specific Functionality")
class car:
   def __init__(self):
      self.engine=Engine()
   def m2(self):
      print("car using Engine class Functionality")
      print(self.engine.a)
      print(self.engine.b)
      self.engine.m1()
c=car()
```

```
c.m2()
```

output:

```
car using Engine class Functionality
20
30
Engine Specific Functionality
```

program:

```
class car:
  def __init__(self,name, model, color):
    self.name=name
    self.model=model
    self.color=color
  def getinfo(self):
    print("car Name:{},Model:{} and color:{}".format(self.name,self.model,self.color))
class Employee:
  def __init__(self,ename,eno,car):
    self.ename=ename
    self.eno=eno
    self.car=car
  def empinfo(self):
    print("Employee Name:", self.ename)
    print("Employee Name:", self.eno)
    print("Employee car info:")
    self.car.getinfo()
c=car("innova","3.5v", 'Blue')
e=Employee("sangam",100000, c)
e.empinfo()
```

output:

```
Employee Name: sangam
Employee Name: 100000
Employee car info:
car Name:innova,Model:3.5v and color:Blue
```

- in the above program employee class Has-a car reference and hence employee class can access all members of car class

program:

```
class x:
  a=30
  def __init__(self):
    self.b=40
  def m1(self):
    print("m1 method of x class")
class y:
  c=30
  def __init__(self):
    self.d=50
  def m2(self):
    print("m2 method of y class")
```

```
    def m3(self):
      x1=x()
      print(x1.a)
      print(x1.b)
      x1.m1()
      print(y.c)
      print(self.d)
      self.m2()
      print("m3 method of y class")
y1=y()
y1.m3()
```

output:

```
30
40
m1 method of x class
30
50
m2 method of y class
m3 method of y class
```

2) By inheritance (IS-A Relationship):

- what ever variables, methods and constructor available in the parent class by default available to the child classes and we are not required to rewrite hence the main advantage of inheritance i code reusability and we can extend existing functionality with some more extra functionality.

syntax: class childclass(parentclass)

Example:

```
class p:
  a=10
  def __init__(self):
    self.b=20
  def m1(self):
    print('parent instance method')
  @classmethod
  def m2(cls):
    print('parent class method')
  @staticmethod
  def m3():
    print("parent static method")
class c(p):
  pass
r=c()
print(r.a)
print(r.b)
r.m1()
r.m2()
r.m3()
```

output:

```
10
20
parent instance method
parent class method
parent static method
```

```
class p:
  10 methods
class c(p):
   5 methods
```

- in the above example parent class contains 10 method and these methods automatically Reusability
- hence child class contains 5 methods

Note: whatever members present in parent class are by default available to the child class through inheritance.

Example:

```
class p:
   def m1(self):
      print("parent class method")
class c(p):
   def m2(self):
      print("child class method")
r=c()
r.m1()
r.m2()
```

output:

```
parent class method
child class method
```

- whatever methods present in parent class are automatically available to the child class and hence on the child class reference we can call both parent class methods and child class methods.
- similarly variable also

Example:

```
class p:
   a=20
   def __init__(self):
      self.b=30
class c(p):
   c=40
   def __init__(s):
      super().__init__() #===>Line-1
      s.d=50
r=c()
print(r.a,r.b,r.c,r.d)
```

output:

```
20 30 40 50
```

- if we comment line-1 then variable b is not available to the child class.

➢ program for inheritance:

```
class person:
    def __init__(s,name,age):
        s.name=name
        s.age=age
    def eatndrink(s):
        print('Eat Biryani and Drink Beer')
class Employee(person):
    def __init__(s,name,age,eno,esal):
        super().__init__(name,age)
        s.eno=eno
        s.esal=esal
    def work (s):
        print("coding python is very easy just like drinking child Beer")
    def empinfo(self):
        print("Employee Name:", self.name)
        print("Employee Age:", self.age)
        print("Employee Number:", self.eno)
        print("Employee Salary:", self.esal)
e=Employee("Sangam Kumar", 39,100,3000000)
e.eatndrink()
e.work()
e.empinfo()
```

output:

```
Eat Biryani and Drink Beer
coding python is very easy just like drinking child Beer
Employee Name: Sangam Kumar
Employee Age: 39
Employee Number: 100
Employee Salary: 3000000
```

❖ IS-A vs HAS-A Relationship:

- if we want to extend existing functionality with some extra functionality then we should go for IS-A Relationship.

Example:

- Employee class extends person class functionality but Employee class just uses car functionality but not extending.

Example:

```
class car:
    def __init__(s,name, model, color):
        s.name=name
        s.model=model
        s.color=color
    def getinfo(s):
```

```
        print("\t car name:{} \n\t model:P{} \n\t color:{}".format(s.name,s.model,s.color))
class person:
    def __init__(s,name,age):
        s.name=name
        s.age=age
    def eatndrink(s):
        print("eat biryani and drink beer")

class Employee(person):
    def __init__(s,name,age,eno,esal,car):
        super().__init__(name, age)
        s.eno=eno
        s.esal=esal
        s.car=car
    def work(self):
        print("python is very easy")
    def empinfo(self):
        print("Employee Name:", self.name)
        print("Employee Age:", self.age)
        print("Employee Number:", self.eno)
        print("Employee Salary:", self.esal)
        print("Employee car info:")
        self.car.getinfo()
c=car("Innova", '3.5V','Blue')
e=Employee("Sangam Kumar", 39,100,3000000,c)
e.eatndrink()
e.work()
e.empinfo()
```

output:

```
eat biryani and drink beer
python is very easy
Employee Name: Sangam Kumar
Employee Age: 39
Employee Number: 100
Employee Salary: 3000000
Employee car info:
        car name:Innova
        model:P3.5V
        color:Blue
```

- in the above example Employee class extends person class functionality but just car class functionality.

Composition vs Aggregation:

- without existing container object if there is no chance of existing contained object then the container and contained objects are strongly associated and that strong association is nothing but composition.

Example:

- University contains several departments and without existing university objects there is no chance of existing department object. hence university and department objects are strongly associated and this strong association is nothing but composition.

❖ Aggregation:

- without existing container object if there is a chance of existing contained object then the container and contained objects are weakly associated and that weak association is nothing but aggregation.

Example:

- department contains several professors. without existing department still there may be a chance of existing professor. hence department and professor objects are weakly associated which is nothing but aggregation.

Example:
```
class student:
    collegeName="DKSRA"
    def __init__(s,name):
        s.name=name
print(student.collegeName)
s=student("Sangam Kumar")
print(s.name)
```
output:
```
DKSRA
Sangam Kumar
```

- in the above example without existing student object there is no chance of existing his name. hence student object and his name are strongly associated which is nothing but composition.
- but without existing student object there may be a chance of existing collegeName. hence student object and collegeName are weakly associated which is nothing but aggregation.

Conclusion:

- the relation between object and its instance variables is always composition whereas the relation between object and static variables is aggregation.

Note: whenever we are creating child class object then child class constructor will be executed. if the child class does not contain constructor, then parent class constructor will be executed, but parent object won't be created.

Example:
```
class p:
    def __init__(self):
        print(id(self))
class c(p):
    pass
r=c()
print(id(r))
```
output:
```
1448461546208
1448461546208
```

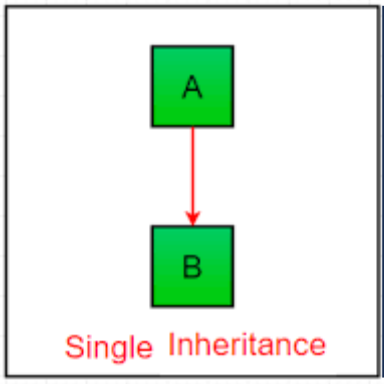
What is inheritance ?:

- Inheritance in Python is a mechanism that allows a class (known as the child or subclass) to inherit properties and behaviors (attributes and methods) from another class (known as the parent or superclass).
- Inheritance enables you to create a new class that is a modified or specialized version of an existing class, inheriting its attributes and methods while adding or overriding them as needed.

❖ Types of inheritance:

1. 1.single inheritance.
2. multiple inheritance
3. hierarchical inheritance
4. multilevel inheritance
5. 5.Hybrid inheritance
6. 6.Cyclic inheritance

1-Single Inheritance



What is Single Inheritance?
Single Inheritance means a **child class** (also called a **derived class**) inherits from **only one parent class** (also called a **base class**).
The child class can use the **properties and methods** of the parent class.

◇ **Why use it?**
- To **reuse code** from the parent class.
- To **extend** or **customize** the behavior of the parent class.

Example: # Define the parent class A (Base class)

```
class A:
    # Constructor method to initialize values a and b
    def __init__(s, a, b):
        s.a = a
        s.b = b
    def disp(self): # Method to display value of a
        print(self.a)
class B(A): # Define child class B that inherits from class A
    def disp1(self): # Method to display both a and b
        print("Number", self.a, self.b)
obj = B(10, 20)  # Create an object of child class B
obj.disp() # Call method from parent class to print value of a
obj.disp1() # Call method from child class to print a and b
```

Output:

```
10
Number 10 20
```

Syntax:

```
# Parent class
class Parent:
    def show(self):
        print("This is the Parent class")
# Child class (inherits from Parent)
class Child(Parent):
    def display(self):
        print("This is the Child class")
# Create object of Child class
obj = Child()
# Access method from Parent class
obj.show()

# Access method from Child class
obj.display()
```

Task-1: - User Login System

```
class User:
    def __init__(self, username, email):
        self.username = username
        self.email = email
    def get_user_info(self):
        print(f"Username: {self.username}")
        print(f"Email: {self.email}")
# Child class (inherits from User)
class AdminUser(User):
    def show_admin_panel(self):
        print(f"Welcome Admin {self.username} to the admin panel.")
# Take input from the user
username = input("Enter admin username: ")
email = input("Enter admin email: ")
# Create object of AdminUser with user input
admin = AdminUser(username, email) # Call method from parent class
admin.get_user_info() # Call method from child class
admin.show_admin_panel()
```

Output:

Enter admin username: admin123
Enter admin email: admin@example.com

Output:

Username: admin123
Email: admin@example.com
Welcome Admin admin123 to the admin panel.

Question based on single inheritance

1. **Student Management System**
   Create a base class Student with details like name and roll number. Create a child class Marks that stores and displays marks of 3 subjects.
2. **Employee Payroll System**
   Define a class Employee with employee name and ID. Inherit it into a class Salary that takes and displays basic salary and calculates gross salary.
3. **Library System**
   Create a class Book that contains book name and author. Inherit it into a class IssuedBook that adds issue date and return date, and displays full book status.
4. **Vehicle Registration System**
   Define a class Vehicle that stores vehicle number and type. Create a child class Owner that stores owner's name and address and displays all details.
5. **Online Shopping Cart**
   Create a class Product with product name and price. Inherit it into a class OrderItem that adds quantity and calculates total price.
6. **Banking System**
   Create a class Account with account number and holder's name. Inherit it into a class Transaction that stores deposit and withdrawal details and prints the balance.

Superclass and super () method

What is a Superclass?

- A **superclass** (also called **base class** or **parent class**) is the class whose properties and methods are inherited by another class.
- In your code, class A is the **superclass**.
- class B is the **child class** (also called **subclass**), which inherits from class A.

What is super()?

- The super() method is used in the **child class** to call the **constructor or methods** of the **parent class**.
- It helps avoid writing the parent class name explicitly.
- It is **cleaner** and follows **good coding practices**, especially when there are **multiple inheritance levels**.

Example-1:

```
class A:  # Base class / Parent class
   def __init__(s, a, b):
      s.a = a
      s.b = b
   def disp(self):
      print(s.a, s.b)
class B(A):  # Child class
   def __init__(s, a, b, c):
      # Calling constructor of parent class using its name (not using super here)
      A.__init__(s, a, b)  # You can also use: super().__init__(a, b)
      s.c = c
   def dis(s):
      print(f"The attributes of base class A - {s.a}, {s.b}")
      print(f"The attributes of child class B - {s.c}")

ob = B(10, 20, 30)
ob.dis()
```

Output:

```
The attributes of base class A - 10, 20
The attributes of child class B - 30
```

How to Use super() Instead of A.__init__()?

Replace this:

```
A.__init__(s, a, b)
```

With:

```
super().__init__(a, b)
```

This is recommended because:

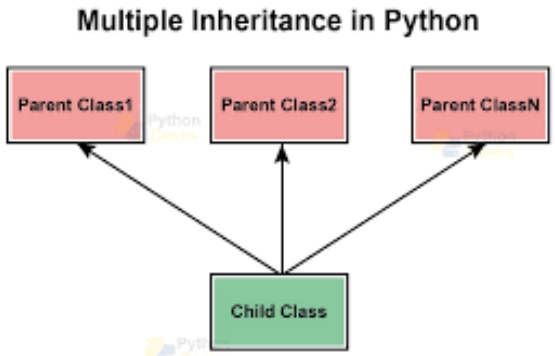
- It automatically finds the right parent class in complex inheritance.
- It's more maintainable and avoids hardcoding the class name.

Final Notes:

- **super()** is mostly used inside **__init__()** to inherit attributes.
- It can also be used to call parent methods like: **super().some_method().**

❖ Multiple Inheritance:

Diagram:



What is Multiple Inheritance?

- **Multiple Inheritance** means a **child class** can inherit from **more than one parent class**.
- This allows the child class to use **methods and attributes from multiple classes**.

Example-1: Syntax of Multiple Inheritance

```
class Father:#Parent class-1
   def __init__(s,fname):
      s.fname=fname
      super()
class Mother:#Parent class-2
   def __init__(s,mname):
      s.mname=mname
class Child(Father,Mother):#Child class
   def __init__(s,fname,mname,child):
      super().__init__(fname)
      Mother.__init__(s,mname)
      s.child=child
   def dis(s):
      print(f"Attributes of Parent class-1 {s.fname}")
      print(f"Attributes of Parent class-2 {s.mname}")
      print(f"Attributes of Child class {s.child}")
ob=Child("Charlie","Ms Charlie","John")
ob.dis()
Child.mro()
```

Output:

```
Attributes of Parent class-1 Charlie
Attributes of Parent class-2 Ms Charlie
Attributes of Child class John
[__main__.Child, __main__.Father, __main__.Mother, object]
```

Example-2: # Parent class 1

```
class A:
   def showA(self):
      print("This is class A")

# Parent class 2
class B:
   def showB(self):
      print("This is class B")

# Child class (inherits from A and B)
class C(A, B):
   def showC(self):
      print("This is class C")

# Create object of child class
obj = C()
obj.showA()   # From class A
obj.showB()   # From class B
obj.showC()   # From class C
```

Note: -

- Multiple inheritance allows **reusing code** from more than one class.
- Python handles it with **Method Resolution Order (MRO)** to avoid conflicts.
- It's useful when a class needs features from **different sources**.

Example-1 School Management System:

```
# Parent class 1
class Student:
    def __init__(self, name, roll):
        self.name = name
        self.roll = roll

    def display_student(self):
        print(f"Student Name: {self.name}")
        print(f"Roll Number: {self.roll}")

# Parent class 2
class Sports:
    def __init__(self, sport_name):
        self.sport_name = sport_name

    def display_sport(self):
        print(f"Sport Chosen: {self.sport_name}")

# Child class
class SchoolRecord(Student, Sports):
    def __init__(self, name, roll, sport_name):
        Student.__init__(self, name, roll)
        Sports.__init__(self, sport_name)

    def display_record(self):
        self.display_student()
        self.display_sport()

# Create object and call
record = SchoolRecord("Tanmay", 101,
"Football")
record.display_record()
```

Output:

```
Student Name: Tanmay
Roll Number: 101
```

Example-2 Employee Attendance + Salary Management System:

```
# Parent class 1: Employee Information
class Employee:
    def __init__(self, name, emp_id):
        self.name = name
        self.emp_id = emp_id
    def display_employee(self):
        print(f"Employee Name: {self.name}")
        print(f"Employee ID: {self.emp_id}")
# Parent class 2: Attendance Information
class Attendance:
    def __init__(self, days_present):
        self.days_present = days_present
    def display_attendance(self):
        print(f"Days Present: {self.days_present}")
# Child class: Inherits from both Employee and Attendance
class EmployeeRecord(Employee, Attendance):
    def __init__(self, name, emp_id, days_present,
salary_per_day):
        Employee.__init__(self, name, emp_id)
        Attendance.__init__(self, days_present)
        self.salary_per_day = salary_per_day
    def calculate_salary(self):
        total_salary = self.days_present * self.salary_per_day
        print(f"Total Salary: ₹{total_salary}")
    def display_record(self):
        self.display_employee()
        self.display_attendance()
        self.calculate_salary()
# Create object and call methods
record = EmployeeRecord("Rajesh", 2001, 26, 800)
record.display_record()
```

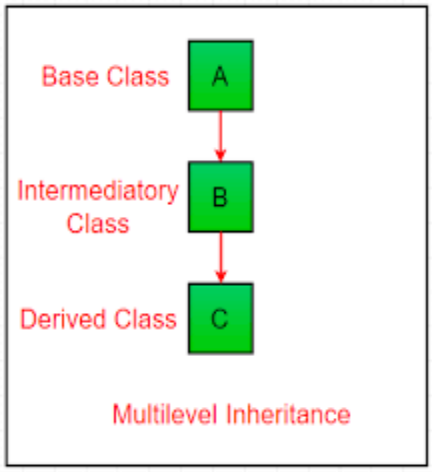
Output:

```
Employee Name: Rajesh
Employee ID: 2001
Days Present: 26
Total Salary: ₹20800
```

1. **Hospital Management System**
   Create a Patient class with patient details and a Doctor class with doctor details. Create a Treatment class that inherits from both and shows full treatment info.
2. **School Report Card System**
   Create a Student class with name and roll number, and a Marks class with subject marks. Create a ReportCard class that inherits from both and calculates total and percentage.
3. **Flight Reservation System**
   Define a Passenger class with passenger info and a FlightDetails class with flight info. Create a Booking class that inherits from both and shows complete booking details.
4. **E-commerce Order Management**
   Create a Customer class with customer info, and a Product class with product name and price. Then create an Order class that inherits from both and shows the order summary and total amount.

❖ Multi-level Inheritance:

Diagram:



Multi-level Inheritance means a class is derived from a **child class**, which is itself derived from **another class**.
It forms a **chain of inheritance**.

Grandparent (Base Class)

↓

Parent (Child of Base)

↓

Child (Child of Parent)

Example:

```
class GrandFather:#Parent class
    def __init__(s,son):
        s.son=son
class Father(GrandFather):#Intermediate
    def __init__(s,son,grandson):
        super().__init__(son)
        s.grandson=grandson
class Son(Father):#Child class
    def __init__(s,son,grandson,child):
        super().__init__(son,grandson)
        s.child=child
    def dis(s):
        print(f"The attributes of Parent
class:{s.son}")
        print(f"The attributes of Intermediate
class:{s.grandson}")
        print(f"The attributes of Child
class:{s.child}")
ob = Son("Charlie","John","Johnson")
ob.dis()
```

output: The attributes of Parent class:Charlie
The attributes of Intermediate class:John
The attributes of Child class:Johnson

Syntax: # Base class

```
class A:
    def methodA(self):
        print("This is class A")

# Derived class (inherits from A)
class B(A):
    def methodB(self):
        print("This is class B")

# Derived class (inherits from B)
class C(B):
    def methodC(self):
        print("This is class C")

# Create object of class C
obj = C()
obj.methodA()  # from A
obj.methodB()  # from B
obj.methodC()  # from C
```

Example-2 Bank Account System # Base class

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

    def show_person(self):
        print(f"Name: {self.name}")
# Intermediate class
class BankAccount(Person):
    def __init__(self, name, account_no):
        super().__init__(name)
        self.account_no = account_no

    def show_account(self):
        print(f"Account Number: {self.account_no}")
# Derived class
class SavingsAccount(BankAccount):
    def __init__(self, name, account_no, balance):
        super().__init__(name, account_no)
        self.balance = balance

    def show_balance(self):
        print(f"Current Balance: ₹{self.balance}")
# Create object of the most derived class
acc = SavingsAccount("Tanmay", 123456789,
5000)
# Call methods from all levels
acc.show_person()    # From Person
acc.show_account()   # From BankAccount
acc.show_balance()   # From SavingsAccount
```

Example-3: Education System

```
# Base class
class Person:
    def __init__(self, name):
        self.name = name
    def show_person(self):
        print(f"Name: {self.name}")
# Intermediate class
class Student(Person):
    def __init__(self, name, student_id):
        super().__init__(name)
        self.student_id = student_id
    def show_student(self):
        print(f"Student ID: {self.student_id}")
# Derived class
class CollegeStudent(Student):
    def __init__(self, name, student_id, college_name, course):
        super().__init__(name, student_id)
        self.college_name = college_name
        self.course = course
    def show_college_info(self):
        print(f"College: {self.college_name}")
        print(f"Course: {self.course}")
# Create object of the most derived class
student = CollegeStudent("Anjali", "CS102", "ABC Engineering
College", "B.Tech Computer Science")
# Call methods from all levels
student.show_person()      # From Person class
student.show_student()     # From Student class
student.show_college_info()  # From CollegeStudent class
```

Note:

- Multi-level inheritance allows a class to build upon features of another **step-by-step**.
- You can use super() to call parent class constructors.
- It helps create a **structured hierarchy** in OOP.

Question

🔷 **1. Library Membership System**

Create a base class Person to store name and age.Create a class Member that inherits from Person and adds membership ID.Then create a Borrower class that adds book title and due date, and displays the full borrowing details.

🔷 **2. Vehicle Tracking System**

Create a class Vehicle with basic info like model and number.Inherit it into Driver class to add driver's name and license number.Then inherit into TripDetails class to record distance, start time, and end time. Display the full trip report.

🔷 **3. Online Course Enrollment System**

Create a base class User with user name and email. Create a class Learner that adds enrolled course name.Then create a Progress class that tracks completed modules and shows progress report for user.

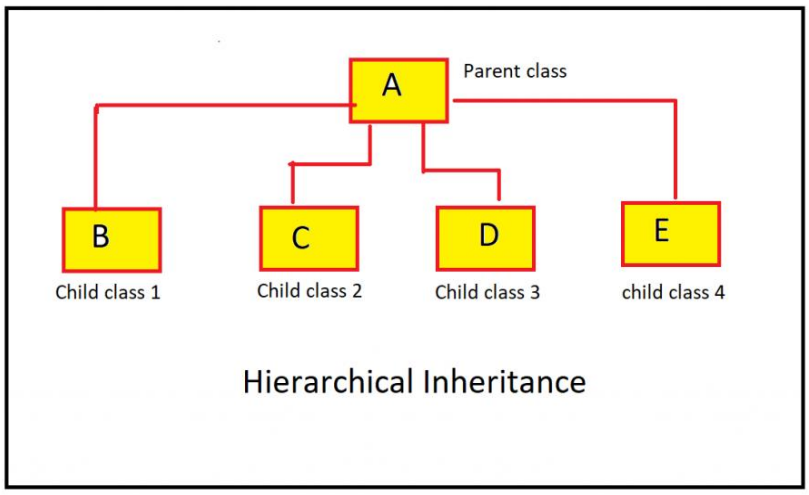
❖ Hierarchical Inheritance:

- What is Hierarchical Inheritance?
- In Hierarchical Inheritance, multiple child classes inherit from a single parent class.
  Think of it like a tree:

```
      Parent
      /  \
  Child1  Child2
```

- Each child gets the features of the same parent but can also have its own features.

Diagram:



Syntax of Hierarchical Inheritance

```
# Parent class
class Parent:
    def display(self):
        print("This is the Parent class")
# Child class 1
class Child1(Parent):
    def show1(self):
        print("This is Child1 class")
# Child class 2
class Child2(Parent):
    def show2(self):
        print("This is Child2 class")
# Create objects of both child classes
obj1 = Child1()
obj1.display()  # from Parent
obj1.show1()    # from Child1

obj2 = Child2()
obj2.display()  # from Parent
obj2.show2()    # from Child2
```

Note:

- One **parent class** can be reused by multiple child classes.
- Each **child class** can have its own unique methods.
- **parent class** methods are **inherited** and accessible in all child classes.

Example:

```
class Parent:#Parent class
    def __init__(s,pname):
        s.pname=pname
class Child1(Parent):#Child class-1
    def __init__(s,pname,cname1):
        super().__init__(pname)
        s.cname1=cname1
    def dis(s):
        print(f"Attributes of Parent class {s.pname}")
        print(f"Attributes of Child class-1 {s.cname1}")
class Child2(Parent):#Child class-2
    def __init__(s,pname,cname2):
        super().__init__(pname)
        s.cname2=cname2
    def dis(s):
        print(f"Attributes of Parent class {s.pname}")
        print(f"Attributes of Child class-2 {s.cname2}")
ob1=Child1("John","Charlie")
print(Child1.mro())
ob1.dis()
ob2=Child2("John","Harry")
print(Child2.mro())
ob2.dis()
```

Output:

```
[<class '__main__.Child1'>, <class '__main__.Parent'>, <class 'object'>]
Attributes of Parent class John
Attributes of Child class-1 Charlie
[<class '__main__.Child2'>, <class '__main__.Parent'>, <class 'object'>]
Attributes of Parent class John
Attributes of Child class-2 Harry
```

Example-2: School Management System using Hierarchical Inheritance

We will create a base class Person, and then two child classes:

- Student (inherits from Person)
- Teacher (inherits from Person)

```
# Base class
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def show_basic_info(self):
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
# Child class 1
class Student(Person):
    def __init__(self, name, age, student_id, grade):
```

```
        super().__init__(name, age)
        self.student_id = student_id
        self.grade = grade
    def show_student_info(self):
        print(f"Student ID: {self.student_id}")
        print(f"Grade: {self.grade}")
# Child class 2
class Teacher(Person):
    def __init__(self, name, age, subject, employee_id):
        super().__init__(name, age)
        self.subject = subject
        self.employee_id = employee_id
    def show_teacher_info(self):
        print(f"Employee ID: {self.employee_id}")
        print(f"Subject: {self.subject}")
# Create object of Student
student = Student("Ankit", 16, "S102", "10th")
print("Student Details:")
student.show_basic_info()
student.show_student_info()

print("\n-------------------------\n")

# Create object of Teacher
teacher = Teacher("Mrs. Sharma", 35, "Mathematics", "T501")
print("Teacher Details:")
teacher.show_basic_info()
teacher.show_teacher_info()
```

- **Output:**

```
Student Details:
Name: Ankit
Age: 16
Student ID: S102
Grade: 10th

-------------------------

Teacher Details:
Name: Mrs. Sharma
Age: 35
Employee ID: T501
Subject: Mathematics
```

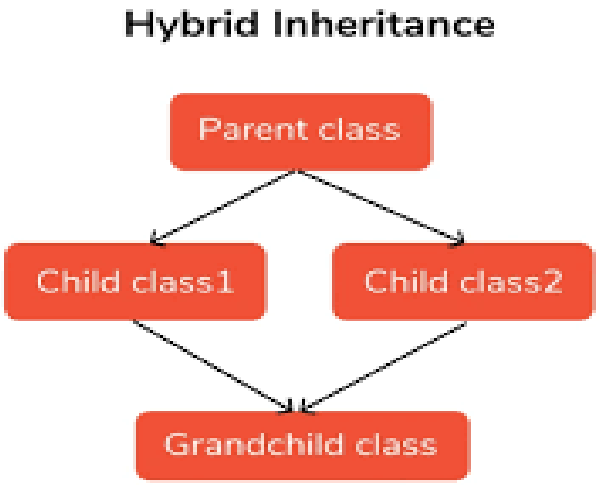
Concept Recap:

- Person is the **parent class**.
- Student and Teacher are **child classes** that inherit from Person.
- This is an example of **Hierarchical Inheritance** — one parent, multiple children.

❖ Hybrid inheritance:

- combination of single, multi-level, multiple, and hierarchical inheritance is known as hybrid inheritance.

Diagram:



Example:

```
# single
        class Grandfather: #base class
          def __init__(s,gname):
            s.gname=gname
          def disp(s):
            print("Grandfather Name is {s.gname}")
        class Father(Grandfather):#Child class for grandfather & base class for child1 & child2
          def __init__(s,gname,fname):
            super().__init__(gname)
            s.fname=fname
          def dis1(s):
            print("Grandfather Name is {s.gname}\n Father name is {s.fname}")
```

Hierarchical:

```
        class child1(Father):#child class
          def __init__(s,gname,fname,s1):
            super().__init__(gname,fname)
            s.s1=s1
          def dis1(s):
            print(f"Grandfather name is {s.gname},Father name is {s.fname},child1 name is {s.s1}")
        class child2(Father):#child class
          def __init__(s,gname,fname,s2):
            super().__init__(gname,fname)
            s.s2=s2
          def dis2(s):
            print(f"Grandfather name is {s.gname},Father name is {s.fname},child2 name is {s.s2}")
        print("single inheritance (garandfather --->father)")
```

```
obj=Grandfather("Rama")
obj.disp()
obj1=Father("GodFather", "Ram")
obj1.dis1()
print("Hiearchical inheritance (father --->child1,father --->child2 )")
obj2=child1("GodFather", "Ramu","Ravi")
obj3=child2("GodFather", "Mohan","Sohan")
obj2.dis1()
obj3.dis2()
```

output:

```
single inheritance (garandfather --->father)
Grandfather Name is {s.gname}
Grandfather Name is {s.gname}
 Father name is {s.fname}
Hiearchical inheritance (father --->child1,father --->child2 )
Grandfather name is GodFather,Father name is Ramu,child1 name is Ravi
Grandfather name is GodFather,Father name is Mohan,child2 name is Sohan
```

Example-2: University System using Hybrid Inheritance

```
# Base class
class Person:
    def __init__(self, name):
        self.name = name
    def show_name(self):
        print(f"Name: {self.name}")
# Single Inheritance: Student inherits from Person
class Student(Person):
    def __init__(self, name, student_id):
        super().__init__(name)
        self.student_id = student_id

    def show_student_id(self):
        print(f"Student ID: {self.student_id}")

# Another child of Person (Hierarchical Inheritance)
class Teacher(Person):
    def __init__(self, name, subject):
        super().__init__(name)
        self.subject = subject

    def show_subject(self):
        print(f"Subject: {self.subject}")

# Multi-level Inheritance: GraduateStudent -> Student -> Person
class GraduateStudent(Student):
    def __init__(self, name, student_id, research_topic):
        super().__init__(name, student_id)
        self.research_topic = research_topic
```

```
    def show_research(self):
        print(f"Research Topic: {self.research_topic}")

# Another class
class Sports:
    def __init__(self, sport_name):
        self.sport_name = sport_name

    def show_sport(self):
        print(f"Sport: {self.sport_name}")

# Multiple Inheritance: Scholar inherits from GraduateStudent and Sports
class Scholar(GraduateStudent, Sports):
    def __init__(self, name, student_id, research_topic, sport_name):
        GraduateStudent.__init__(self, name, student_id, research_topic)
        Sports.__init__(self, sport_name)

    def show_scholar_details(self):
        self.show_name()
        self.show_student_id()
        self.show_research()
        self.show_sport()

# Create object of Scholar (Hybrid Inheritance)
scholar = Scholar("Tanmay", "ST101", "AI in Education", "Chess")
print(" Scholar Details:")
scholar.show_scholar_details()

print("\n Teacher Details:")
teacher = Teacher("Dr. Meena", "Data Science")
teacher.show_name()
teacher.show_subject()
```

🔹 **Output:**

```
Scholar Details:
Name: Tanmay
Student ID: ST101
Research Topic: AI in Education
Sport: Chess

Teacher Details:
Name: Dr. Meena
Subject: Data Science
```

✅ **Summary:**

- Person is the base class (used for hierarchical, single, and multi-level inheritance).
- Scholar inherits from both GraduateStudent (multi-level) and Sports (multiple inheritance).
- This setup forms **Hybrid Inheritance**.

Question based on inheritance

1. 🔹 **Employee Management System**
   Create a Person class and extend it into Employee. Then create different departments like Manager, Developer using **hierarchical inheritance** and display their information.
2. 🔹 **E-commerce System**
   Create a User class and derive Customer and Seller classes from it. Add product and order details using **hierarchical inheritance**.
3. 🔹 **Hospital Management System**
   Design a system using Person as a base class, inherited by Doctor, Patient, and Nurse. Implement **hierarchical inheritance** and show their duties and data.
4. 🔹 **University Admission System**
   Use **multi-level inheritance** to model Person → Student → CollegeStudent. Add data like name, age, student ID, course, and show admission info.
5. 🔹 **Banking System**
   Use **multiple inheritance** with classes AccountHolder and BankPolicy, and a derived class BankCustomer that inherits from both. Show customer info and policy rules.
6. 🔹 **Transport Booking System**
   Create a Vehicle class and derive Bus and Train classes. Use **hierarchical inheritance** to display ticket details and travel mode.
7. 🔹 **Smart Home System**
   Create devices like Light, Fan, and SmartDevice, and use **multiple inheritance** to create a SmartRoom class that controls all devices.
8. 🔹 **Online Course Platform**
   Create a base class User, and extend it into Instructor and Learner. Use **hierarchical and multi-level inheritance** to manage course creation and progress tracking.
9. 🔹 **Library Management System**
   Design classes: Person → Member → PremiumMember using **multi-level inheritance**. Add methods to show membership type and borrowed books.
10. 🔹 **Hybrid Project: Research Scholar System**

- Create classes Person, Student, GraduateStudent, and Researcher using **hybrid inheritance** (multi-level + multiple). Include attributes like research topic and supervisor.

Theoratical Question and answer

1. What is inheritance in Python?

- Inheritance allows a class (child) to access properties and methods of another class (parent).

2. What is single inheritance?

- Single inheritance means one child class inherits from one parent class.

3. What is multiple inheritance?

- Multiple inheritance means a child class inherits from two or more parent classes.

4. What is multi-level inheritance?

- Multi-level inheritance means a class inherits from a class which itself inherits from another class.

5. What is hierarchical inheritance?

- Hierarchical inheritance means multiple child classes inherit from a single parent class.

6. What is hybrid inheritance?

- Hybrid inheritance is a combination of two or more types of inheritance (like single + multiple).

7. What is the use of super() in inheritance?

- super() is used to call the parent class's constructor or methods from the child class.

8. Can a child class override parent methods?

- Yes, a child class can override methods defined in the parent class.

9. What is the benefit of inheritance?

- Inheritance promotes code reuse and supports modular and scalable program design.

10. How do you define a child class in Python?

- By writing: class Child(Parent): where Parent is the parent class.

11. What keyword is used to define a class in Python?

- The class keyword is used to define a class in Python.

12. What is a base class in inheritance?

- A base class is the parent class from which other classes inherit.

13. What is a derived class in inheritance?

- A derived class is the child class that inherits from a base (parent) class.

14. Can constructors be inherited in Python?

- Yes, constructors can be inherited using super() or by calling the parent class explicitly.

15. How can we access parent class attributes in child class?

- By using super() or ParentClassName.__init__(self, args) in the child class.

16. What happens if both parent and child have the same method name?

- The child class method overrides the parent class method.

17. Is multiple inheritance allowed in Python?

- Yes, Python supports multiple inheritance.

18. How do you check the method resolution order (MRO) in Python?

- By using ClassName.__mro__ or help(ClassName).

19. What is the difference between super() and class name while calling parent constructor?

- super() is preferred for maintainability and automatic MRO; ClassName.__init__ is manual.

20. Can private attributes be inherited in Python?

- Private attributes (with __) are not directly inherited but can be accessed using name mangling.

21. What is method overriding in Python inheritance?

Method overriding is redefining a parent class method in the child class.

22. Can you inherit from multiple classes in Python at the same time?

Yes, Python allows multiple inheritance by separating parent classes with commas.

23. What is the default base class of all classes in Python?

All classes in Python inherit from the built-in object class by default.

24. How do you prevent a class from being inherited?

Python does not directly support final classes, but you can raise an exception in the constructor.

25. Can we call parent methods in a child class?

Yes, by using super().method_name() or ParentClass.method_name(self).

26. What is the benefit of multi-level inheritance?

It helps create a class hierarchy and reuse features across multiple layers.

27. Can we inherit both data members and methods from a parent class?

Yes, both data members (variables) and methods are inherited.

28. What happens if a child class doesn't have an __init__() method?

The parent class's __init__() method will be called automatically if available.

29. What is meant by MRO (Method Resolution Order)?

MRO is the order in which classes are searched for a method during inheritance.

30. How do you inherit only specific methods from a class?

Python doesn't allow partial inheritance; you must override or ignore unwanted methods manually.

Method Resolution Order (MRO)

What is Method Resolution Order (MRO) in Python?

- **MRO** is the **order in which Python looks for methods or attributes** in a class hierarchy (especially when multiple inheritance is used).
- It tells Python which class to search **first, second, third, etc.** when calling a method.

◆ **Where is MRO used?**

MRO is mainly used in:

- **Multiple Inheritance**
- **Hybrid Inheritance**

Key Points to Remember

Term	Explanation
MRO	Method Resolution Order – order in which Python checks classes.
C3 Algorithm	Algorithm used to find MRO.
DLR	Depth-Left-to-Right search.
Left class gets priority	When classes are listed in inheritance, leftmost is checked first.
Semuele Pedroni	The person who proposed the C3 algorithm.
mro()	A built-in method to see the MRO of a class.

◆ **Syntax**

```
print(ClassName.mro())
# OR
help(ClassName)
```

Example 1: Simple Multiple Inheritance

```
class A:
    def show(self):
        print("Class A")

class B(A):
    def show(self):
        print("Class B")

class C(A):
    def show(self):
        print("Class C")

class D(B, C):
    pass

d = D()
d.show()  # Output?
print(D.mro())
```

Output:

```
Class B
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>,
<class 'object'>]
```

Explanation:

- Python checks **D → B → C → A**
- Since B has show(), it uses that and stops.

Example 2: Hybrid Inheritance (Real MRO Case)

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

print(D.mro())
```

Output:

[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]

How MRO Works: The Merge Rule

MRO(D) = D + merge(MRO(B), MRO(C), [B, C])

Let's break it:

- MRO(B) = [B, A, object]
- MRO(C) = [C, A, object]
- Merge = Start comparing **head of each list** (first element) and add it to MRO **if it's not in the tail of any other list**.

Step by Step:

1. Take B (head of MRO(B)), it's not in the tail of any list → Add it
2. Now C is the next head → Add it
3. Next is A → Add it
4. Then object → Add it

Final: [D, B, C, A, object]

Real-Life Analogy

Think of MRO like **asking a question in a classroom** with **multiple teachers**:

- You ask your main teacher first (left-most parent),
- If they don't know, you go to the next one,
- This continues until someone gives you the answer.

Summary

- MRO helps Python decide **which method to call** when **multiple classes are involved**.
- Uses **C3 Linearization (DLR rule)**.
- Left class gets **higher priority**.
- Use .mro() or help(ClassName) to check MRO.
- Important in **multiple and hybrid inheritance**.

MRO-- METHOD RESOLUTION ORDER

- The root class or by default the base class is OBJECT class

Example:

```
class A:#Base class
   def __init__(s,a,b):
      s.a=a
      s.b=b
class B(A):#Child class
   def __init__(s,a,b,c):
      #Inheriting the base class properties by super/class name
      #super().__init__(a,b)
      A.__init__(s,a,b)
      s.c=c
   def dis(s):
      print(f"The attributes of base class A- {s.a},{s.b}")
      print(f"The attributes of child class B-{s.c}")
ob=B(10,20,30)
ob.dis()
B.mro()
dir(B)
dir(object)
dir(int)#To display all the inbuilt methods of a class use dir()
```

output:

```
The attributes of base class A- 10,20
The attributes of child class B-30
['__abs__',
 '__add__',
 '__and__',
 '__bool__',
 '__ceil__',
 '__class__',
 '__delattr__',
 .
 .
 .
 .

 'from_bytes',
 'imag',
 'numerator',
 'real',
 'to_bytes']
        [<class '__main__.x'>, <class '__main__.a'>, <class '__main__.b'>, <class 'object'>]
        [<class '__main__.y'>, <class '__main__.b'>, <class '__main__.c'>, <class 'object'>]
        [<class '__main__.p'>, <class '__main__.x'>, <class '__main__.a'>, <class
        '__main__.y'>, <class '__main__.b'>, <class '__main__.c'>, <class 'object'>]
```

problem:

- **Create a user choice-based Calculator with OOP concept**

```
class Calculator:
   def __init__(s,num1,num2):#attributes are num1 & num2
      s.num1=num1
      s.num2=num2
   def add(s):
      print(f"Addition = {s.num1+s.num2}")
   def sub(s):
      print(f"Subtraction = {abs(s.num1-s.num2)}")
   def mul(s):
      print(f"Multiplication = {s.num1*s.num2}")
   def div(s):
      print(f"Division = {s.num1/s.num2}")
n1=int(input("Number-1 "))
n2=int(input("Number 2- "))
ob = Calculator(n1,n2)
print("Enter your choice:-\n1 for Addition\n2 for Subtraction\n3 for Multiplication\n4 for
Division")
ch=int(input("Choice: "))
if ch==1:
   ob.add()
elif ch==2:
   ob.sub()
elif ch==3:
   ob.mul()
elif ch==4:
   ob.div()
else:
   print("invalid input..")
```

output:

```
Number-1 10
Number 2- 20
Enter your choice:-
1 for Addition
2 for Subtraction
3 for Multiplication
4 for Division
Choice: 1
Addition = 30
```

problem:

- Create a class SuperList which will inherit the inbuilt class list
- Also the methods like append,pop has to be implemented.
- display the final output

Example:

```
class ListOfLists(list):#Here list is the parent class & SuperList is child class
   def dis(s):#user defined method
```

```
        print("The length of the list is-",len(s))
        print("The list is-",s)
n=int(input("Length:-"))
ob=ListOfLists()
for i in range(n):
   ob.append(int(input()))
ob.dis()
ob.pop()
ob.dis()
print(type(ob))
print(dir(ob))
help(list)#help is used to display the structure of class
a=(1,2,3,4)#a is the object of class tuple
print(type(a))
```

output:

```
Length:-3
5
6
7
The length of the list is- 3
The list is- [5, 6, 7]
The length of the list is- 2
The list is- [5, 6]
<class '__main__.ListOfLists'>
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',
'__delitem__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__module__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'append', 'clear', 'copy', 'count', 'dis', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |   Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
```

```
|  Data and other attributes defined here:
|
|  __hash__ = None

<class 'tuple'>
```

mro example:

```
class a:
   def m1(self):
      print("a class method")
class b:
   def m1(self):
      print("b class method")
class c:
   def m1(self):
      print("c class method")
class x(a,b):
   def m1(self):
      print("x class method")
class y(b,c):
   def m1(self):
      print("y is method")
class p(x,y,c):
   def m1(self):
      print("p class method")
p=p()
p.m1()
x=x()
x.m1()
r=c()
r.m1()
```

output:

```
p class method
x class method
c class method
```

note: in the above example p class m1() method will be considered. if p class does not contain m1() method then as per MRO, x class method will be considered. if x class does not contain then a class method will be considered and this process will be continued.

the method resolution in the following order: pxaybco

Example-3 for MRO

Example

```
mro(o)=object
mro(d)=d,object
mro(e)=e,object
mro(f)=f,object
mro(b)=b,d,f,object
mro(c)=c,d,f,object
mro(a)=a+merge(mro(b),mro(c),bc)
```

super () method

- super () is a built-in function which is useful to call the super class constructors, variables and methods from the child class.

What is super() method in Python?

- The super() method is a **built-in function** used in **inheritance** to call the **parent (super) class's constructor, methods, or variables** from a **child class**.

Syntax:

```
super().method_name()
super().__init__(arguments)
```

Why use super()?

- To reuse the parent class's code
- Makes the code cleaner and avoids repetition
- Useful especially in inheritance and multiple inheritance

Example:

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

    def show(self):
        print("Name:", self.name)

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)  # Calling parent constructor
        self.age = age

    def show(self):
        super().show()        # Calling parent method
        print("Age:", self.age)

c = Child("Tanmay", 16)
c.show()
```

Output:

```
Name: Tanmay
Age: 16
```

In Short:

- super() helps access parent class features.
- Used in child class to **call** parent methods or constructors easily.
- Promotes **code reuse** and **clean design** in object-oriented programming.

Example 1: Call Parent Constructor and Method Using super()

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

    def show_info(self):
```

```
        print("Name:", self.name)
        print("Age:", self.age)

    class Student(Person):
      def __init__(self, name, age, roll, marks):
        super().__init__(name, age)  # calling parent constructor
        self.roll = roll
        self.marks = marks

      def show_info(self):
        super().show_info()  # calling parent method
        print("Roll:", self.roll)
        print("Marks:", self.marks)

    s = Student("Tanmay", 16, 101, 95)
    s.show_info()
```

Output:

```
Name: Tanmay
Age: 16
Roll: 101
Marks: 95
```

Example-2: - Accessing All Parent Methods Using super()

```
class Parent:
  a = 30
  def __init__(self):
    self.b = 9

  def m1(self):
    print("Instance method")
  @classmethod
  def m2(cls):
    print("Class method")
  @staticmethod
  def m3():
    print("Static method")
class Child(Parent):
  def __init__(self):
    super().__init__()
    print(super().a)
    super().m1()
    super().m2()
    super().m3()
obj = Child()
```

Output:

```
30
Instance method
Class method
Static method
```

Example:

```
class p:
  def __init__(s,name,age):
    s.name=name
    s.age=age
  def disp(s):
    print("Name:",s.name)
    print("Age:",s.age)
class student(p):
  def __init__(s,name,age,rollno,marks):
    super().__init__(name,age)
    s.rollno=rollno
    s.marks=marks
  def disp(s):
    super().disp()
    print("Roll No:",s.rollno)
    print("marks:",s.marks)
s1=student("Sangam Kumar",18,103,99)
s1.disp()
```

output:

```
Name: Sangam Kumar
Age: 18
Roll No: 103
marks: 99
```

- in the above program we are using super () method to call parent class constructor and display() method

Example-3:Calling Parent Variable, Constructor, and Method Using super()

```
class Parent:
    a = 100  # Class variable (static variable)

    def __init__(self):
        self.b = 200  # Instance variable
        print("Parent constructor called")

    def parent_method(self):
        print("Parent instance method called")

    @classmethod
    def class_method(cls):
        print("Parent class method called")

    @staticmethod
    def static_method():
        print("Parent static method called")


class Child(Parent):
    def __init__(self):
        super().__init__()          # Calling Parent constructor
        print("Child constructor called")

        print("Accessing parent variable using super():", super().a)  # Class variable
        super().parent_method()       # Calling Parent method
        super().class_method()        # Calling Parent class method
        super().static_method()       # Calling Parent static method


# Create object of child class
child_obj = Child()
```

Output:

```
Parent constructor called
Child constructor called
Accessing parent variable using super(): 100
Parent instance method called
Parent class method called
Parent static method called
```

Example: Library Management System

```
# Parent class
class Library:
    library_name = "Central City Library"  # Class variable

    def __init__(self, book_list):
        self.books = book_list  # Instance variable
        print("Library system initialized.")

    def display_books(self):
        print("Available books in", self.library_name + ":")
        for book in self.books:
            print("-", book)

    @classmethod
    def show_library_name(cls):
        print("Library Name (class method):", cls.library_name)

    @staticmethod
    def library_rules():
        print("Rule: Return books within 15 days.")

# Child class
class Member(Library):
    def __init__(self, name, book_list):
        super().__init__(book_list)  # Call parent constructor
        self.name = name
        print(f"Member '{self.name}' added.")

        # Accessing parent class variable
        print("Accessing library name using super():", super().library_name)
        # Calling parent methods
        super().display_books()
        super().show_library_name()
        super().library_rules()
# Creating object of Member
member1 = Member("Tanmay", ["Python Basics", "Data Science", "Machine Learning"])
```

Example:

```
Library system initialized.
Member 'Tanmay' added.
Accessing library name using super(): Central City Library
Available books in Central City Library:
- Python Basics
- Data Science
- Machine Learning
Library Name (class method): Central City Library
Rule: Return books within 15 days.
```

1) super(d,self).m1()

it will call m1() method of super class of d.

1) a.m1(self)

it will call a class m1() method

2) super(d,self).m1()

Example:

```
class a:
   def m1(s):
      print("a class method")
class b(a):
   def m1(s):
      print("b class method")
class c(b):
   def m1(s):
      print("c class method")
class d(c):
   def m1(s):
      print("d class method")
class e(d):
   def m1(s):
      a.m1(s)
r=e()
r.m1()
r1=c()
r1.m1()
r3=d()
r3.m1()
```

output:

```
a class method
c class method
d class method
```

POLYMORPHISM

- "**Poly**" = many, "**Morph**" = forms
- Polymorphism means "many forms" – same function/operator behaving differently depending on the context or object.
- The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

Example:

- yourself is best example of polymorphism. in front of your parents, you will have one type of behaviour and with friends another type of behaviour same person but different behaviours at different places, which is nothing but polymorphism

Programming Example:

```
print(10 + 20)      # adds numbers
print("Hello" + " World")  # joins strings

print(5 * 3)        # multiplies numbers
print("Hi " * 3)    # repeats string
```

Note:- Same **+** and ***** operators do **different jobs** — **this is operator overloading**, a type of polymorphism.

❖ There are following topics are important

1). Duck typing philosophy of python

2). overloading

1.operator overloading
2.method overloading
3.constructor overloading

3). overriding

1.method overriding
2.constructor overriding

➢ In Python, Polymorphism is achieved by-

1. METHOD OVERRIDING
2. METHOD OVERLOADING
3. OPERATOR OVERLOADING

1. **Duck Typing:-** Based on behavior, not type
2. **Operator Overloading:-** Same operator for different data types
3. **Method Overloading:-** Same method name, different parameters
4. **Method Overriding:-** Child class redefines parent class method
5. **Constructor Overriding:-** Child class constructor replaces parent

Difference between Overloading and Overriding in Python

- **Overloading**: Same method name with different number or type of parameters (handled using default arguments or *args). Happens in the **same class**.
- **Overriding**: Same method name and parameters in **child class** to change behavior of **parent class** method. Happens in **inheritance**.

Basic Difference between Overloading and Overriding in Python:

Feature	Overloading	Overriding
Definition	Same function name, different parameters	Same function name and parameters in child class
Purpose	In the same class	In parent-child (inheritance) relationship
Example Use	Flexibility in function inputs	Custom behavior in subclass

Example of Overloading (using default arguments):

```
class Math:
    def add(self, a, b=0):
        return a + b

m = Math()
print(m.add(5))      # Output: 5
print(m.add(5, 3))   # Output: 8
```

Example of Overriding:

```
class Animal:
    def sound(self):
        print("Animal makes sound")
class Dog(Animal):
    def sound(self):  # Overriding the parent class method
        print("Dog barks")
d = Dog()
d.sound()    # Output: Dog barks
```

1. Method Overriding – Achieved Using Inheritance

```
class A:  # Parent class
    def dis(self):
        print("Class A")
class B(A):  # Child class
    def dis(self):
        print("Class B")
ob = B()
ob.dis()
```

Output: Class B

Explanation: The dis() method in class B overrides the dis() method of class A due to inheritance.

2. Method Overloading – Implemented Using Default Arguments in Python

```
class A:
    def __init__(self, a, b=0, c=3, d=8):
        self.a = a
        self.b = b
        self.c = c
        self.d = d

    def dis(self):
        print(f"The values are:\n{self.a}\n{self.b}\n{self.c}\n{self.d}")

ob1 = A(10)          # 1 argument
ob1.dis()

ob2 = A(10, 20)      # 2 arguments
ob2.dis()

ob3 = A(10, 50, 60)  # 3 arguments
ob3.dis()

ob4 = A(1, 2, 3, 4)  # 4 arguments
ob4.dis()
```

Output:

```
The values are:
10
0
3
8
The values are:
10
20
3
8
The values are:
10
50
60
8
The values are:
1
2
3
4
```

Explanation:

Method overloading is simulated in Python by using default arguments. The __init__ method accepts a variable number of arguments.

1) Duck Typing in Python

- Python is **dynamically typed** – we don't declare types explicitly.
- Type is decided **at runtime** based on behavior, not class.
- Follows: *"If it walks like a duck and talks like a duck, it's a duck."*

Example:

```
class Duck:
   def talk(self): print("Quack... Quack...")
class Dog:
   def talk(self): print("Bow Bow...")
class Goat:
   def talk(self): print("Myaah Myaah...")

def f1(obj):
   obj.talk()

for obj in [Duck(), Dog(), Goat()]:
   f1(obj)
```

Output:

```
Quack... Quack...
Bow Bow...
Myaah Myaah...
```

Problem: If obj has no talk() method → AttributeError.

Solution: Use hasattr() to check before calling:

```
def f1(obj):
   if hasattr(obj, 'talk'):
      obj.talk()
```

2) Overloading in Python

- **Same operator or method used for different purposes**.

Examples:

```
print(3 + 6)          # 9 → Arithmetic
print('skills' + 'hub')  # skillshub → String concatenation

print(3 * 6)          # 18 → Multiplication
print('hi' * 3)       # hihihi → String repetition
```

Method Overloading (Simulated):

```
def deposit(amount):  # Can accept cash, cheque, or dd
   print(f"Deposited: {amount}")
deposit("cash")
deposit("cheque")
deposit("dd")
```

Let me know if you want this as a printable note or slide!

Types of Overloading in Python

- **Overloading in Python** means using the **same name** (for operator or method) to perform **different tasks** based on the context, usually handled using **default arguments** or *args.

There are **3 types of overloading**:

1. **Operator Overloading**
2. **Method Overloading**
3. **Constructor Overloading**

1. Operator Overloading

➞ Same operator used for different purposes.

Built-in Examples:

```
print(3 + 6)        # 9 (Addition)
print('a' + 'b')    # ab (Concatenation)

print(3 * 2)        # 6 (Multiplication)
print('hi' * 3)     # hihihi (Repetition)
```

Custom Class Example:

```
class Book:
   def __init__(self, pages):
      self.pages = pages

   def __add__(self, other):  # Overloading +
      return self.pages + other.pages
b1 = Book(100)
b2 = Book(200)
print("Total pages =", b1 + b2)
```

Output: Total pages = 300

Magic Methods (Few Examples):

Operator	Method
+	__add__()
-	__sub__()
*	__mul__()
>	__gt__()
<=	__le__()

Comparison Example:

```
class Student:
   def __init__(self, name, marks):
      self.name = name
      self.marks = marks
   def __gt__(self, other): return self.marks > other.marks
   def __le__(self, other): return self.marks <= other.marks
s1 = Student("A", 90)
s2 = Student("B", 85)
print(s1 > s2)   # True
print(s1 <= s2)  # False
```

2. Method Overloading

➞ Same method name with different arguments.

Not supported directly in Python – last method will override the previous ones.

Not Working:

```
class Test:
    def m1(self): print("No arg")
    def m1(self, a): print("One arg")
    def m1(self, a, b): print("Two args")
t = Test()
t.m1(10, 20)  # Only last method works
```

How to achieve it: Using default or variable arguments

```
class Test:
    def sum(self, a=None, b=None, c=None):
        if a and b and c:
            print("Sum:", a + b + c)
        elif a and b:
            print("Sum:", a + b)
        else:
            print("Need 2 or 3 numbers")
t = Test()
t.sum(10, 20)      # 30
t.sum(10, 20, 30)  # 60
```

With *args:

```
class Test:
    def sum(self, *nums):
        print("Sum =", sum(nums))
t = Test()
t.sum(10, 20)
t.sum(10, 20, 30)
```

3. Constructor Overloading

Not supported directly. Python only considers the **last __init__()**.

Not Working:

```
class Test:
    def __init__(self): print("No-arg")
    def __init__(self, a): print("One-arg")
    def __init__(self, a, b): print("Two-arg")
t = Test(10, 20)  # Only last constructor works
```

Use default arguments:

```
class Test:
    def __init__(self, a=None, b=None, c=None):
        print("Constructor with 0, 1, 2, or 3 args")
t1 = Test()
t2 = Test(10)
t3 = Test(10, 20)
t4 = Test(10, 20, 30)
```

With *args:

```
class Test:
    def __init__(self, *args):
        print("Constructor with variable args")

t1 = Test()
t2 = Test(10)
t3 = Test(10, 20)
t4 = Test(1, 2, 3, 4)
```

3) Overriding

❖ What is Overriding?

- **Overriding** means **redefining a method or constructor in the child class** that already exists in the parent class.
  It is mainly used in **inheritance** to change or extend the behavior of parent class methods.

1. Method Overriding

- If a **child class** has a method with the **same name** as in the **parent class**, the **child class version is used**.
- Used to **customize behavior**.

Syntax & Example-1:

```
class Parent:
    def show(self):
        print("This is Parent")

class Child(Parent):
    def show(self):
        print("This is Child")  # Overriding
method

c = Child()
c.show()  # Output: This is Child
```

Example-2:

```
class Animal:
    def sound(self):
        print("Some animal sound")

class Dog(Animal):
    def sound(self):
        print("Bark")

d = Dog()
d.sound()  # Output: Bark
```

Example-3:

```
class Vehicle:
    def wheels(self):
        print("4 wheels")

class Bike(Vehicle):
    def wheels(self):
        print("2 wheels")

b = Bike()
b.wheels()  # Output: 2 wheels
```

Example-4:

```
class Shape:
    def area(self):
        print("Calculating area...")

class Circle(Shape):
    def area(self):
        print("Area = πr²")

c = Circle()
c.area()  # Output: Area = πr²
```

2. Constructor Overriding

- When a **child class defines its own __init__()**, it **overrides** the parent class constructor.
- The parent constructor is **not called automatically**, you must call it manually if needed.

Syntax & Example:

```
class Parent:
    def __init__(self):
        print("Parent Constructor")
class Child(Parent):
    def __init__(self):
        print("Child Constructor")  # Overrides parent constructor
c = Child()  # Output: Child Constructor
```

To call Parent constructor too:

```
class Child(Parent):
    def __init__(self):
        super().__init__()   # Calls parent constructor
        print("Child Constructor")
c = Child()
```

Output: Parent Constructor
Child Constructor

Example-3: Constructor Overriding with Arguments

```
class Parent:
    def __init__(self, name):
        print("Parent constructor:", name)

class Child(Parent):
    def __init__(self, name):
        print("Child constructor:", name)

c = Child("Python")
# Output: Child constructor: Python
```

Example-4:

```
class Parent:
    def __init__(self):
        print("Parent constructor")

class Child(Parent):
    def __init__(self):
        print("Child constructor")

c = Child()
# Output: Child constructor
```

Example-3: Constructor Overriding with super()

```
class Parent:
    def __init__(self):
        print("Parent constructor")

class Child(Parent):
    def __init__(self):
        super().__init__()  # Call parent constructor
        print("Child constructor")

c = Child()
# Output:
# Parent constructor
# Child constructor
```

Example-4: Constructor Overriding with Different Arguments

```
class Parent:
    def __init__(self, age):
        print("Parent constructor, Age:", age)

class Child(Parent):
    def __init__(self, age, school):
        print("Child constructor, Age:", age,
"School:", school)

c = Child(12, "Greenwood High")
# Output: Child constructor, Age: 12 School:
Greenwood High
```

Example: Student Project - Constructor Overriding

```
# Parent class
class Person:
    def __init__(self, name, age):
        print("Person Constructor")
        self.name = name
        self.age = age

# Child class
class Student(Person):
    def __init__(self, name, age, course):
        # Overriding the constructor and calling parent using super()
        super().__init__(name, age)
        print("Student Constructor")
        self.course = course

    def display(self):
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"Course: {self.course}")

# Taking input from the user
name = input("Enter student name: ")
age = int(input("Enter student age: "))
course = input("Enter course name: ")

# Creating Student object
s1 = Student(name, age, course)
s1.display()
```

Output:

```
Enter student name: Ankit
Enter student age: 20
Enter course name: B.Tech
Person Constructor
Student Constructor
Name: Ankit
Age: 20
Course: B.Tech
```

What's Happening:

- Student class overrides Person's constructor.
- super().__init__() is used to call the parent constructor.
- User input is taken for a real-world scenario like a student registration system.

Example: Bank Account – Method Overloading using Default Arguments

```
class BankAccount:
    def __init__(self, holder, balance=0):
        self.holder = holder
        self.balance = balance

    # Overloaded method using default arguments
    def deposit(self, cash=None, cheque=None):
        if cash is not None:
            self.balance += cash
            print(f"Deposited ₹{cash} cash")
        if cheque is not None:
            self.balance += cheque
            print(f"Deposited ₹{cheque} cheque")
        print(f"Total Balance: ₹{self.balance}")

# Taking input from user
name = input("Enter account holder name: ")
acc = BankAccount(name)

print("\nChoose deposit type:\n1. Cash\n2. Cheque\n3. Both")
choice = input("Enter your choice: ")

if choice == "1":
    amount = int(input("Enter cash amount: "))
    acc.deposit(cash=amount)
elif choice == "2":
    amount = int(input("Enter cheque amount: "))
    acc.deposit(cheque=amount)
elif choice == "3":
    cash_amt = int(input("Enter cash amount: "))
    cheque_amt = int(input("Enter cheque amount: "))
    acc.deposit(cash=cash_amt, cheque=cheque_amt)
else:
    print("Invalid choice!")
```

Output:

```
Enter account holder name: Priya
Choose deposit type:
1. Cash
2. Cheque
3. Both
Enter your choice: 3
Enter cash amount: 1000
Enter cheque amount: 3000
Deposited ₹1000 cash
Deposited ₹3000 cheque
Total Balance: ₹4000
```

- The deposit method acts differently based on arguments provided — simulating **method overloading**.
- You can deposit cash, cheque, or both.
- This is a real-world use case of overloading behavior in Python.

ABSTRACT METHOD

What is Abstraction?

- Abstraction means **hiding the details** and showing only the important parts.
- For example, when you drive a car, you use the steering wheel, accelerator, and brake — you don't worry about how the engine works inside.

Abstract Method

- A method **without body (no code)** is called an **abstract method**.
- It just says **"This method must exist, but I'll write its code later."**
- Declared using @abstractmethod decorator.
- You must **import** it from the abc module.

Syntax:

```
from abc import ABC, abstractmethod
class MyClass(ABC):  # abstract class
    @abstractmethod
    def my_method(self):
        pass
```

Abstract Class

- A class that contains at least **one abstract method**.
- You **can't create an object** of an abstract class directly.
- Use it as a **blueprint** for other classes.

Examples

Example 1: Basic Abstract Class

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        print("Bark!")

class Cat(Animal):
    def sound(self):
        print("Meow!")

d = Dog()
d.sound()   # Output: Bark!
c = Cat()
c.sound()   # Output: Meow!
```

Example 2: Vehicle Abstract Class

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def wheels(self):
        pass

class Car(Vehicle):
    def wheels(self):
        return 4

class Bike(Vehicle):
    def wheels(self):
        return 2

c = Car()
print(c.wheels())  # Output: 4

b = Bike()
print(b.wheels())  # Output: 2
```

Example 3: Abstract Shape

```
from abc import ABC, abstractmethod
    class Shape(ABC):
        @abstractmethod
        def area(self):
            pass

    class Circle(Shape):
        def area(self):
            return 3.14 * 5 * 5

    class Square(Shape):
        def area(self):
            return 4 * 4

    print(Circle().area())  # Output: 78.5
    print(Square().area())  # Output: 16
```

Example: Payment System

```
from abc import ABC, abstractmethod

class Payment(ABC):
    @abstractmethod
    def make_payment(self, amount):
        pass

class CreditCard(Payment):
    def make_payment(self, amount):
        print(f"Paid ₹{amount} using Credit Card.")

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

# Create objects of child classes
card = CreditCard()
upi = UPI()

# Make payments
card.make_payment(1500)
 # Output: Paid ₹1500 using Credit Card.
upi.make_payment(500)
 # Output: Paid ₹500 using UPI.
```

Note:

What's Happening:

- Payment is an **abstract class** – it defines the **structure**.
- make_payment() is an **abstract method** – no implementation in base class.
- CreditCard and UPI are **child classes** that implement the method in **different ways**.
- This is useful in real projects where different types of objects must follow a **common rule or interface**.

Conclusion

- Use @abstractmethod when you want to **enforce method rules** in child classes.
- An abstract class is like a **template**.
- You **can't create an object** of an abstract class directly.
- All abstract methods must be implemented in child classes before using.

Let me know if you want examples for __str__() vs repr() or public/private/protected!

- In simple terms, abstraction in Python allows you to focus on what an object does rather than how it does it, making it easier to work with and understand complex systems.
    - abstract class
    - interface
    - public private and protected members
    - __str__() method
    - difference between str() and repr() function

1. Abstract Class

- An **abstract class** is like a blueprint. You **can't create objects** of it directly. It's used when you want to **force child classes** to define some methods.

Example:

```
from abc import ABC, abstractmethod
class Animal(ABC):  # abstract class
   @abstractmethod
   def sound(self):
      pass

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

d = Dog()
print(d.sound())  # Output: Bark
```

2. Interface

- Python doesn't have a special interface keyword like Java, but we can **use abstract classes with only abstract methods** as interfaces.

Example:

```
from abc import ABC, abstractmethod
class Vehicle(ABC):  # acts like interface
   @abstractmethod
   def start(self):
      pass

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

c = Car()
c.start()  # Output: Car started
```

3. Public, Private, and Protected Members

- public: accessible everywhere.
- protected (_single underscore): by convention, should be accessed only within class or subclass.
- private (__double underscore): cannot be accessed directly from outside the class.

Example:

```
class Demo:
   def __init__(self):
```

```
        self.public = "I am Public"
        self._protected = "I am Protected"
        self.__private = "I am Private"

d = Demo()
print(d.public)      # OK
print(d._protected)  # Possible, but not recommended
# print(d.__private)  # Error
print(d._Demo__private)  # Accessing private using name mangling
```

4. __str__() Method:- Used to return a **user-friendly string** when you print an object.

Example:

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

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

        s = Student("Tanmay")
        print(s)  # Output: Student Name: Tanmay
```

5. Difference between str() and repr()

Function	Purpose	Output
str()	User-friendly string	Easy to read
repr()	Developer-friendly (debug)	Exact/formal

Example:

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

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

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

        b = Book("Python Basics")
        print(str(b))   # Output: Book title: Python Basics
        print(repr(b))  # Output: Book('Python Basics')
```

CONCREATE CLASS VS ABSTRACT CLASS VS INTERFACE

1. **Interface:**
   If we **don't know anything about the implementation** and only have the **requirement specifications**, then we should go for an **interface** (in Python, this is represented using a fully abstract class with only abstract methods).
2. **Abstract Class:**
   If we are **partially implementing the logic**, i.e., **some methods have implementation and some don't**, then we should use an **abstract class**.
3. **Concrete Class:**
   If we are providing the **complete implementation** and are ready to **provide the service**, then we should go for a **concrete class**.

Example in Python:

```
from abc import ABC, abstractmethod
# Interface equivalent: Fully abstract class
class CollegeAutomation(ABC):
    @abstractmethod
    def m1(self):
        pass
    @abstractmethod
    def m2(self):
        pass
    @abstractmethod
    def m3(self):
        pass
# Abstract class: Partial implementation
class AbsCls(CollegeAutomation):
    def m1(self):
        print("m1 method implementation")

    def m2(self):
        print("m2 method implementation")

# Concrete class: Complete implementation
class ConcreteCls(AbsCls):
    def m3(self):
        print("m3 method implementation")

# Creating object of concrete class and calling methods
r = ConcreteCls()
r.m1()
r.m2()
r.m3()
```

Output:

```
m1 method implementation
m2 method implementation
m3 method implementation
```

PUBLIC, PROTECTED AND PRIVATE ATTRIBUTES

1. Public Attributes

- **By default, every attribute is public.**
- You can access public attributes **from anywhere** — inside the class or outside the class.
- **No special symbol** is needed.

Example: name = "skills"

2. Protected Attributes

- A **protected attribute** is meant to be accessed **within the class** and **by its child classes only**.
- Declared by prefixing the attribute with a **single underscore _**.

Syntax: _name = "skills"

- **this is just a convention**, not real protection — you can still access it from outside the class.

3. Private Attributes

- A **private attribute** can be accessed **only within the class**.
- Declared by prefixing the attribute with **double underscores __**.
- It gets **name mangled**, meaning its real name changes internally (e.g., __z becomes _ClassName__z).

Syntax: __name = "skills"

Example Program (Simple and Clear)

```
class Example:
    public_attr = "I am Public"
    _protected_attr = "I am Protected"
    __private_attr = "I am Private"
    def show_attributes(self):
        print("Inside class:")
        print(self.public_attr)
        print(self._protected_attr)
        print(self.__private_attr)
# Create object
obj = Example()
# Accessing attributes from inside class
obj.show_attributes()
print("\nOutside class:")
# Public attribute - accessible
print(obj.public_attr)
# Protected attribute - accessible (not recommended)
print(obj._protected_attr)
# Private attribute - not accessible directly
# print(obj.__private_attr)  #  This will raise an AttributeError
# Access private attribute using name mangling (not recommended)
print(obj._Example__private_attr)
```

Output:

```
Inside class:
I am Public
I am Protected
I am Private

Outside class:
I am Public
I am Protected
I am Private
```

Access Level	Prefix	Access Scope	Notes
Public	*(none)*	Everywhere	Default access
Protected	_	Within class and subclasses	Just a **convention**, not enforced
Private	__	Only within the class (name mangled)	Real restriction (with name mangling)

❖ How to access private variables from outside of the class:

1. Accessing Private Variables from Outside the Class

Key Point:

- You **cannot directly** access private variables declared with __ (double underscore) from outside the class.
- But in Python, you **can still access them indirectly** using **name mangling**:
- object_reference._ClassName__variable_name

Example 1: Accessing Private Variable

```
class Test:
    def __init__(self):
        self.__x = 30  # private variable

# Create object
r = Test()

# Access private variable using name mangling
print(r._Test__x)
```

Output: 30

Why this works?

- Python internally changes __x to _Test__x, so you can access it with that name.

2. __str__() Method in Python

Key Point:

- When you print an object, Python calls the __str__() method automatically.
- By default, it returns a message like:
- <__main__.ClassName object at 0x000001B4D3C8A2B0>
- To show a **meaningful message**, we override the __str__() method in our class.

Example 2: Custom __str__() for Student Class

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

    def __str__(self):
        return "This is student with Name: {} and Roll No: {}".format(self.name, self.rollno)

# Create objects
s1 = Student("Sangam Kumar", 103)
s2 = Student("Satyam Kumar", 106)

# Print the objects
print(s1)
print(s2)
```

Output:

```
This is student with Name: Sangam Kumar and Roll No: 103
This is student with Name: Satyam Kumar and Roll No: 106
```

Difference between str() repr()
or
Difference between __str__() and __repr__()

Difference between str() and repr()

Feature	str()	repr()
Calls Method	Calls __str__()	Calls __repr__()
Purpose	For **readability** (user-friendly)	For **debugging** and **developer clarity** (unambiguous)
Output	Informal string	Formal string (can be used to recreate the object)
Usage Example	print(obj)	repr(obj) or just type obj in console
Recover Object	Usually **cannot** convert back	Often **can** convert back using eval()

- **Example 1: DateTime Object**

```
import datetime
today = datetime.datetime.now()
# Using str()
s1 = str(today)
print("str(today):", s1)

# Using repr()
s2 = repr(today)
print("repr(today):", s2)

# Trying to recreate object using eval()
# This works only with repr()
d = eval(s2)
print("After eval(repr):", d)
```

Output:

```
str(today): 2025-05-29 14:45:22.123456
repr(today): datetime.datetime(2025, 5, 29, 14, 45, 22, 123456)
After eval(repr): 2025-05-29 14:45:22.123456
```

str() gives a nice date-time string, but you **can't convert it back** using eval().
repr() gives a format that can be **evaluated back** to the original object.

Example 2: Custom Class with __str__() and __repr__()

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

    def __str__(self):
        return f"Student(Name: {self.name}, Roll No: {self.rollno})"

    def __repr__(self):
        return f"Student('{self.name}', {self.rollno})"
```

```
s = Student("Tanmay", 101)

# str() calls __str__()
print("Using str():", str(s))

# repr() calls __repr__()
print("Using repr():", repr(s))

# Reconstructing object using eval() and repr()
s2 = eval(repr(s))
print("Recreated Object:", s2)
```

Output:

```
Using str(): Student(Name: Tanmay, Roll No: 101)
Using repr(): Student('Tanmay', 101)
Recreated Object: Student(Name: Tanmay, Roll No: 101)
```

Final Notes:

- Use str() or __str__() for **end-user display**.
- Use repr() or __repr__() for **debugging, logging, or saving object states**.
- **repr() is preferred** when you may need to **reconstruct** the object later (e.g., with eval()).

Mini Project-1: Project Name: Banking Application

Program:

```
class Account:
    def __init__(self,name,balance,min_balance):
        self.name=name
        self.balance=balance
        self.min_balance=min_balance
    def deposit(self,amount):
        self.balance+=amount
    def withdraw(self, amount):
        if self.balance-amount>=self.min_balance:
            self.balance-=amount
        else:
            print("sorry, Insufficient Balance")
    def printstatment(self):
        print("Account Balance:",self.balance)
class Current(Account):
    def __init__(self,name,balance):
        super().__init__(name,balance,min_balance=1000)
    def __str__(self):
        return "{}'s current account with balance:{} ".format(self.name,self.balance)
class saving(Account):
    def __init__(self,name,balance):
        super().__init__(name, balance,min_balance=0)
    def __str__(self):
        return "{}'s Saving account with balance:{} ".format(self.name,self.balance)
```

```
r=saving("Sangam Kumar",1000000)
print(r)
r.deposit(5000)
r.printstatment()
r.withdraw(4000)
r.withdraw(5000)
print(r)

c=Current("Sataym Kumar",30000)
c.deposit(2000)
print(c)
c.withdraw(3300)
print(c)
```

output:

```
Sangam Kumar's Saving account with balance:1000000
Account Balance: 1005000
Sangam Kumar's Saving account with balance:996000
Sataym Kumar's current account with balance:32000
Sataym Kumar's current account with balance:28700
```

Mini Project: Library Management System

- Manage books in a library: borrowing, returning, and checking availability.

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

    def __str__(self):
        status = "Available" if self.is_available else "Checked Out"
        return f"'{self.title}' by {self.author} - {status}"

class Library:
    def __init__(self, name):
        self.name = name
        self.books = []

    def add_book(self, book):
        self.books.append(book)
        print(f"Book '{book.title}' added to the library.")

    def show_books(self):
        print(f"\nBooks in {self.name} Library:")
        for book in self.books:
            print(book)

    def borrow_book(self, title):
```

```
        for book in self.books:
            if book.title.lower() == title.lower():
                if book.is_available:
                    book.is_available = False
                    print(f"You have successfully borrowed '{book.title}'.")
                    return
                else:
                    print(f"Sorry, '{book.title}' is already borrowed.")
                    return
        print(f"Book '{title}' not found in the library.")

    def return_book(self, title):
        for book in self.books:
            if book.title.lower() == title.lower():
                if not book.is_available:
                    book.is_available = True
                    print(f"You have successfully returned '{book.title}'.")
                    return
                else:
                    print(f"'{book.title}' was not borrowed.")
                    return
        print(f"Book '{title}' not found in the library.")

# Create library
my_library = Library("City Central")

# Add books
b1 = Book("The Alchemist", "Paulo Coelho")
b2 = Book("1984", "George Orwell")
b3 = Book("Clean Code", "Robert C. Martin")

my_library.add_book(b1)
my_library.add_book(b2)
my_library.add_book(b3)

# Show all books
my_library.show_books()

# Borrow books
my_library.borrow_book("1984")
my_library.borrow_book("1984")  # Already borrowed

# Return a book
my_library.return_book("1984")
my_library.return_book("The Alchemist")  # Not borrowed

# Final state
my_library.show_books()
```

Mini Project-2: Enhanced Banking Application

Upgrades Made:

- Added unique **account numbers**.
- Track and **display transaction history**.
- Print detailed **account statement**.
- Use of __str__() and __repr__() for better object display.
- Improved naming and formatting.
- Error handling with user-friendly messages.

Program:

```
import datetime
class Account:
    account_counter = 1001  # Static variable for unique account numbers
    def __init__(self, name, balance, min_balance):
        self.name = name
        self.balance = balance
        self.min_balance = min_balance
        self.acc_no = Account.account_counter
        Account.account_counter += 1
        self.transactions = []  # To store transaction history
        self._record_transaction("Account Opened", balance)
    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            self._record_transaction("Deposit", amount)
        else:
            print(" Deposit amount must be positive.")
    def withdraw(self, amount):
        if amount <= 0:
            print(" Withdrawal amount must be positive.")
        elif self.balance - amount >= self.min_balance:
            self.balance -= amount
            self._record_transaction("Withdraw", amount)
        else:
            print(" Sorry, Insufficient Balance.")
    def _record_transaction(self, type, amount):
        time = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        self.transactions.append(f"{time} | {type}: ₹{amount} | Balance: ₹{self.balance}")
    def print_statement(self):
        print(f"\n Statement for {self.name} (A/C No: {self.acc_no})")
        for txn in self.transactions:
            print(txn)
        print(f"Final Balance: ₹{self.balance}\n")
    def __str__(self):
        return f"{self.name} | A/C No: {self.acc_no} | Balance: ₹{self.balance}"
    def __repr__(self):
        return f"Account('{self.name}', {self.balance}, {self.min_balance})"
class Current(Account):
```

```
    def __init__(self, name, balance):
        super().__init__(name, balance, min_balance=1000)
    def __str__(self):
        return f"{self.name}'s Current Account (A/C No: {self.acc_no}) | Balance:
₹{self.balance}"
class Saving(Account):
    def __init__(self, name, balance):
        super().__init__(name, balance, min_balance=0)
    def __str__(self):
        return f"{self.name}'s Saving Account (A/C No: {self.acc_no}) | Balance: ₹{self.balance}"
# ==== Testing the Program ====
# Saving Account
s1 = Saving("Sangam Kumar", 1000000)
print(s1)
s1.deposit(5000)
s1.withdraw(4000)
s1.withdraw(5000)
s1.print_statement()
# Current Account
c1 = Current("Satyam Kumar", 30000)
print(c1)
c1.deposit(2000)
c1.withdraw(3300)
c1.withdraw(27000)  # Should fail due to min_balance
c1.print_statement()
```

Sample Output:

```
Sangam Kumar's Saving Account (A/C No: 1001) | Balance: ₹1000000
Satyam Kumar's Current Account (A/C No: 1002) | Balance: ₹30000
Statement for Sangam Kumar (A/C No: 1001)
2025-05-29 15:05:10 | Account Opened: ₹1000000 | Balance: ₹1000000
2025-05-29 15:05:10 | Deposit: ₹5000 | Balance: ₹1005000
2025-05-29 15:05:10 | Withdraw: ₹4000 | Balance: ₹1001000
2025-05-29 15:05:10 | Withdraw: ₹5000 | Balance: ₹996000
```

🔸 **Final Balance: ₹996000**

```
Statement for Satyam Kumar (A/C No: 1002)
2025-05-29 15:05:10 | Account Opened: ₹30000 | Balance: ₹30000
2025-05-29 15:05:10 | Deposit: ₹2000 | Balance: ₹32000
2025-05-29 15:05:10 | Withdraw: ₹3300 | Balance: ₹28700
Sorry, Insufficient Balance.
```

🔸 **Final Balance: ₹28700**

Would you like to extend this project further with:

- User login system
- PIN/password protection
- Interest calculation
- Admin dashboard?

Banking Application with dynamic user registration

```
import datetime
class Account:
    account_counter = 1001
    accounts = {}

    def __init__(self, name, balance, min_balance, pin):
        self.name = name
        self.balance = balance
        self.min_balance = min_balance
        self.acc_no = Account.account_counter
        Account.account_counter += 1
        self.pin = pin
        self.transactions = []
        Account.accounts[self.acc_no] = self
        self._record_transaction("Account Opened", balance)

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            self._record_transaction("Deposit", amount)
        else:
            print(" Deposit amount must be positive.")

    def withdraw(self, amount):
        if amount <= 0:
            print("Withdrawal amount must be positive.")
        elif self.balance - amount >= self.min_balance:
            self.balance -= amount
            self._record_transaction("Withdraw", amount)
        else:
            print("Insufficient Balance.")

    def calculate_interest(self, rate, years):
        interest = (self.balance * rate * years) / 100
        self.balance += interest
        self._record_transaction(f"Interest Credited ({rate}% for {years} yr)", interest)

    def _record_transaction(self, type, amount):
        time = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        self.transactions.append(f"{time} | {type}: ₹{amount} | Balance: ₹{self.balance}")

    def print_statement(self):
        print(f"\n Statement for {self.name} (A/C No: {self.acc_no})")
        for txn in self.transactions:
            print(txn)
        print(f" Final Balance: ₹{self.balance}\n")
```

```
    def verify_pin(self, entered_pin):
        return self.pin == entered_pin

    def __str__(self):
        return f"{self.name} | A/C No: {self.acc_no} | Balance: ₹{self.balance}"

    def __repr__(self):
        return f"Account('{self.name}', {self.balance}, {self.min_balance}, PIN hidden)"


class Current(Account):
    def __init__(self, name, balance, pin):
        super().__init__(name, balance, min_balance=1000, pin=pin)

    def __str__(self):
        return f"{self.name}'s Current Account (A/C No: {self.acc_no}) | Balance: ₹{self.balance}"

class Saving(Account):
    def __init__(self, name, balance, pin):
        super().__init__(name, balance, min_balance=0, pin=pin)

    def __str__(self):
        return f"{self.name}'s Saving Account (A/C No: {self.acc_no}) | Balance: ₹{self.balance}"

# Login System
def login(account_number, pin):
    account = Account.accounts.get(account_number)
    if account and account.verify_pin(pin):
        print(f"\nWelcome, {account.name}!")
        return account
    else:
        print(" Invalid account number or PIN.")
        return None

# Admin Dashboard
def admin_dashboard():
    print("\nAdmin Dashboard:")
    for acc_no, acc in Account.accounts.items():
        print(f"A/C No: {acc_no} | Name: {acc.name} | Balance: ₹{acc.balance} | Type: {'Current' if
isinstance(acc, Current) else 'Saving'}")

# Dynamic User Registration
def register_user():
    print("\nRegister New Account")
    name = input("Enter your name: ")
    while True:
        account_type = input("Enter account type (saving/current): ").lower()
```

```
        if account_type in ['saving', 'current']:
            break
        print(" Invalid type! Please enter 'saving' or 'current'.")

    while True:
        try:
            balance = float(input("Enter initial deposit amount: "))
            break
        except ValueError:
            print("Please enter a valid number.")
    while True:
        try:
            pin = int(input("Set a 4-digit PIN: "))
            if 1000 <= pin <= 9999:
                break
            else:
                print("PIN must be 4 digits.")
        except ValueError:
            print("Enter numbers only.")

    if account_type == "saving":
        account = Saving(name, balance, pin)
    else:
        account = Current(name, balance, pin)
    print(f"\nAccount created successfully!\nAccount No: {account.acc_no} | Name: {account.name}")
    return account
# ==== MAIN PROGRAM ====
if __name__ == "__main__":
    print("Welcome to SmartBank App")
    users = []
    for _ in range(2):
        users.append(register_user())

    admin_dashboard()

    while True:
        print("\nLogin to your account")
        try:
            acc_no = int(input("Enter Account Number: "))
            pin = int(input("Enter PIN: "))
            user = login(acc_no, pin)
            if user:
                while True:
                    print("\nChoose Operation:")
                    print("1. Deposit")
                    print("2. Withdraw")
                    print("3. Print Statement")
                    print("4. Calculate Interest")
```

```
            print("5. Logout")
            choice = input("Enter choice (1-5): ")
            if choice == '1':
                amt = float(input("Enter amount to deposit: "))
                user.deposit(amt)
            elif choice == '2':
                amt = float(input("Enter amount to withdraw: "))
                user.withdraw(amt)
            elif choice == '3':
                user.print_statement()
            elif choice == '4':
                rate = float(input("Enter interest rate: "))
                years = float(input("Enter number of years: "))
                user.calculate_interest(rate, years)
                print("Interest added.")
            elif choice == '5':
                print("Logged out.")
                break
            else:
                print("Invalid choice.")
        except ValueError:
            print("Invalid input. Please enter numbers only.")
```

Output:

```
Welcome to SmartBank App
Register New Account
        Enter your name: Sangam
        Enter account type (saving/current): saving
        Enter initial deposit amount: 100000
        Set a 4-digit PIN: 1234
Account created successfully!
Register New Account
        Enter your name: Satyam
        Enter account type (current): current
        Enter initial deposit amount: 50000
        Set a 4-digit PIN: 5678
Account created successfully!
Admin Dashboard:
        A/C No: 1001 | Name: Sangam | Balance: ₹100000 | Type: Saving
        A/C No: 1002 | Name: Satyam | Balance: ₹50000 | Type: Current
```

Encapsulation

What is Encapsulation in Python?

- **Encapsulation** means **wrapping or binding** data (variables) and methods (functions) into a **single unit** (class), and **restricting access** to the internal details of that object.

This is one of the **four main principles of OOP** (Object-Oriented Programming):

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

Why is Encapsulation Important?

- To protect data from **direct access** or **accidental changes**.
- To implement **security** and **control** over who can access or modify data.
- To hide the internal state of objects from the outside world (**data hiding**).

How Encapsulation Works in Python

- Use **classes** to bind variables and methods together.
- Use **private (__var)** and **protected (_var)** attributes to hide data.
- Access data only through **getter/setter methods**.

Example 1: Basic Encapsulation with Private Variable

```
class Student:
    def __init__(self, name, marks):
        self.__name = name        # private variable
        self.__marks = marks      # private variable

    def get_info(self):           # public method to access private data
        return f"Name: {self.__name}, Marks: {self.__marks}"

    def set_marks(self, new_marks): # setter method
        if 0 <= new_marks <= 100:
            self.__marks = new_marks
        else:
            print("Invalid marks!")

s = Student("Tanmay", 85)
print(s.get_info())       # Accessing using method

s.set_marks(95)           # Updating marks safely
print(s.get_info())

# print(s.__marks)        Error: cannot access private variable directly
```

Example 2: Encapsulation in Banking System

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

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

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

    def check_balance(self):
        return f"Available balance: ₹{self.__balance}"

a = BankAccount("Tanmay", 1000)
a.deposit(500)
a.withdraw(300)
print(a.check_balance())    #  Proper access

# print(a.__balance)       Not allowed (private)
```

- **Example 3: Encapsulation with Getter and Setter**

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

    def get_salary(self):
        return self.__salary

    def set_salary(self, new_salary):
        if new_salary >= 0:
            self.__salary = new_salary
        else:
            print("Salary can't be negative!")

emp = Employee("Rajesh", 25000)
print("Initial Salary:", emp.get_salary())
emp.set_salary(30000)
print("Updated Salary:", emp.get_salary())
emp.set_salary(-5000)  # Will print error
```

Term	Meaning
Encapsulation	Wrapping data and methods in a class
Data Hiding	Preventing direct access to class variables
Private Variable	Defined with __ (double underscore)
Getter Method	Used to read private data
Setter Method	Used to update private data with validation

Project Title: Student Report Card System (Using Encapsulation)

Features:

- Add student details (Name, Roll No, Marks)
- Access student info using methods
- Validate marks using setter
- Prevent direct access to internal data

Program:

```
class Student:
    def __init__(self, name, roll_no, math, science, english):
        self.__name = name
        self.__roll_no = roll_no
        self.__math = math
        self.__science = science
        self.__english = english
    # Getters (to access private data)
    def get_name(self):
        return self.__name
    def get_roll_no(self):
        return self.__roll_no
    def get_marks(self):
        return {
            "Math": self.__math,
            "Science": self.__science,
            "English": self.__english
        }
    # Setters (to update marks safely with validation)
    def set_math_marks(self, marks):
        if 0 <= marks <= 100:
            self.__math = marks
        else:
            print("Invalid marks for Math")
    def set_science_marks(self, marks):
        if 0 <= marks <= 100:
            self.__science = marks
        else:
            print("Invalid marks for Science")
    def set_english_marks(self, marks):
        if 0 <= marks <= 100:
            self.__english = marks
        else:
            print("Invalid marks for English")
    # Calculate total and percentage
    def get_total(self):
        return self.__math + self.__science + self.__english
    def get_percentage(self):
        return self.get_total() / 3
```

```
    # Print report card
    def print_report(self):
        print("\n📝 Student Report Card")
        print("-------------------------")
        print(f"Name      : {self.__name}")
        print(f"Roll No   : {self.__roll_no}")
        print("Marks:")
        print(f"  Math    : {self.__math}")
        print(f"  Science : {self.__science}")
        print(f"  English : {self.__english}")
        print(f"Total     : {self.get_total()}")
        print(f"Percentage: {self.get_percentage():.2f}%")
        print("-------------------------\n")
# ---------------------------- # Demo Usage
# Creating student object
student1 = Student("Tanmay Kumar", 101, 85, 90, 80)
student1.print_report()
# Updating marks
student1.set_math_marks(95)
student1.set_science_marks(105)  # Invalid
student1.set_english_marks(88)
# Print updated report
student1.print_report()
```

Output: Student Report Card

```
-------------------------
Name      : Sangam Kumar
Roll No   : 101
Marks:
  Math    : 85
  Science : 90
  English : 80
Total     : 255
Percentage: 85.00%
-------------------------
 Invalid marks for Science
Student Report Card
-------------------------
Name      : Sangam Kumar
Roll No   : 101
Marks:
  Math    : 95
  Science : 90
  English : 88
Total     : 273
Percentage: 91.00%
-------------------------
```

TYPES OF ERRORS

- **In any programming language there are 2 types of errors are possible.**
  1). Syntax Error
  2). Runtime Errors

1) Syntax Errors:

- The errors which occur because of invalid syntax are called syntax errors.

Example:

```
a=20
If a==10
Print("skills")
```

Output: syntaxError:Invalid syntax

Example:

```
Print "skills"
Syntax Error: Missing parentheses in call to print
```

Note:

- programmer is responsible to correct these syntax errors. Once all syntax errors are corrected then only program execution will be started.

2). Runtime Errors:

- Also known as exceptions.
- While executing the program if something goes wrong because of end user input or programming logic or memory problems etc then we will get Runtime Errors.

Example:

1. Print(10/0) → ZeroDivisionError:division by zero
2. Print(10/ "ten")→ TypeError:unsupported operand type(s) for /: 'int' and 'str'
3. a=int(input("Enter the number:"))
   print(a)

output:

```
Enter the number: ten
ValueError:invalid literal for int() with base 10: "ten'
```

Note: Exception handling concept applicable for Runtime Errors but for syntax errors

What is Exceptional Handling?

- An unwanted and unexpected event that disturbs normal flow of program is called exception.

What is Exception Handling in Python?

- **Exception Handling** is a way to **handle errors** that occur **during the execution** of a program (called **runtime errors**) so the program doesn't crash unexpectedly.

Why Use Exception Handling?

Without exception handling:

```
a = int(input("Enter a number: "))
print(10 / a)
```

If the user enters 0, this will crash with:

ZeroDivisionError: division by zero

- But with exception handling, we can show a friendly message instead of crashing.

Basic Syntax of try-except Block

```
try:
    # risky code
except ErrorType:
    # handle the error
```

Example:

```
try:
    a = int(input("Enter a number: "))
    print(10 / a)
except ZeroDivisionError:
    print("❌ Cannot divide by zero.")
except ValueError:
    print("❌ Please enter a valid number.")
```

Keywords in Exception Handling

1. try

- Code that **might cause an error** is written here.

2. except

- Block that handles the error if it occurs.

3. else

- Executes if **no exception** occurs in try block.

4. finally

- Runs **no matter what** (error or no error). Used to clean up resources.

Syntax with else and finally

```
try:
    # code that may raise an exception
except ErrorType:
    # handling code
else:
    # runs only if no exception
finally:
    # always runs
```

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("You can't divide by zero.")
except ValueError:
    print("Invalid input. Please enter a number.")
else:
    print("Division successful:", result)
finally:
    print("This block always runs.")
```

- **Catching Multiple Exceptions**

```
try:
    # risky code
except (ZeroDivisionError, ValueError) as e:
    print("Error:", e)
```

- **Catching All Exceptions (not recommended for most use cases)**

```
try:
    # risky code
except Exception as e:
    print("Something went wrong:", e)
```

- **Raising Custom Exceptions**

You can also raise your own errors using raise.

Example:

```
age = int(input("Enter your age: "))
if age < 18:
    raise ValueError("Age must be 18 or above.")
else:
    print("You are eligible.")
```

Keyword	Purpose
try	Wraps code that may raise error
except	Catches and handles the error
else	Executes if no error occurs
finally	Always executes (clean-up tasks etc.)
raise	Manually trigger an exception

Example:

- NameError
- TypeError
- ValueError
- ZeroDivisionError
- IndexError
- KeyError
- FileNotFoundError
- IOError
- AttributeError
- ImportError
- KeyboardInterrupt
- MemoryError
- ArithmeticError
- AssertionError
- SystemError
- Exception

- **SyntaxError:** Raised when there is a syntax error in your code, such as a missing colon, unmatched parentheses, or invalid indentation.
- **IndentationError:** Raised when there is an issue with the indentation of your code, typically caused by inconsistent use of spaces or tabs.
- **NameError:** Raised when a local or global name is not found. This usually occurs when you try to access a variable or function that does not exist.
- **TypeError:** Raised when an operation is performed on an object of an inappropriate type. For example, trying to add a string and an integer.
- **ValueError:** Raised when an operation receives an argument of the correct type but with an inappropriate value. For example, trying to convert a string that doesn't represent a valid integer to an integer.
- **ZeroDivisionError:** Raised when you try to divide a number by zero.
- IndexError: Raised when you try to access an index that is out of range for a sequence (e.g., a list or a string).
- **KeyError:** Raised when you try to access a dictionary key that does not exist.
- **FileNotFoundError:** Raised when an attempt to open a file fails because the specified file does not exist.
- **IOError:** Raised when an input/output operation fails, such as reading from or writing to a file.
- **AttributeError:** Raised when you try to access an attribute or method of an object that does not have that attribute or method.
- **ImportError:** Raised when there is an issue with importing a module, such as when the module is not found or cannot be loaded.
- KeyboardInterrupt: Raised when the user interrupts the program's execution by pressing Ctrl+C (KeyboardInterrupt signal).
- **MemoryError:** Raised when an operation runs out of memory.
- **ArithmeticError:** The base class for arithmetic exceptions. It includes exceptions like OverflowError, FloatingPointError, and ZeroDivisionError.
- **AssertionError:** Raised when an assert statement fails.
- **SystemError:** Raised when the interpreter detects an internal error.

- Exception: The base class for all built-in exceptions.
  - ➢ It is highly recommended to handle exceptions. The main objective of exception handling is graceful termination of the program (i.e we should not block our resource and we should not miss anything)
  - ➢ Exception handling does not mean repairing exception. We have to define alternative way to continue rest of the program normally.

Example:

- For example, our programming requirement is reading data from remote file locating at Patna. At runtime if Patna file is not available then the program should not be terminated abnormally, we have to provide local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

❖ Default Exception Handling in python:

- Every exception in python is an object. For every exception type the corresponding classes are available.
- Whenever an exception occurs PVM will create the corresponding exception object and will check for handling code. If handling code is not available then python interpreter terminates the program abnormally and prints corresponding exception information to the console.
- The rest of the program won't be executed.

Example:

```
print("skills")
print(10/0)
print('t3')
```

output:

```
ZeroDivisionError
Cell In[1], line 2
     1 print("skills")
----> 2 print(10/0)
     3 print('t3')
ZeroDivisionError: division by zero
```

- every exception in python is a class
- all exception classes are child classes of base Exception. i.e. every exception class extends base Exception either directly or indirectly. hence Base Exception acts as root for python exception hierarchy.
- most of the times being a programmer we have to concentrate exception and its child classes.

❖ customized exception handling by using try-except

- it is highly recommended to handle exception is called risky code and we have to take risky code inside try block. the corroding handling code we have to take inside except block

```
try:
  risky code
except xxx:
  handling code/alternative code
```

- **without try-except:**

Example:

```
print("stmt-1")
print(10/0)
```

```
print("stmt-3")
```

output:

```
stmt-1
ZeroDivisionError: division by zero
```

- **with try-except:**

Example:

```
print("stmt-1")
try:
   print(10/0)
except ZeroDivisionError:
   print(39/3)
print("stmt-3")
```

output:

```
stmt-1
13.0
stmt-3
```

❖ Control flow in try-except:

```
try:
   statment-1
   statment-2
   statment-3
except xyz:
   statment-4
statment-5
```

case-1 if there is no exception

1,2,3,5,6 and normal termination

case-2 if an execept raised at statement-2 and corresponding except block matched

1,4,5,6 normal termination

case-3 if an exception rose at statetment-2 and corresponding except block not matched

1,abnormal termination

case-4: if an exception rose at statement-4 or at statement-5 then it is always abnormal termination

❖ conclusions:

1). within the try block if anywhere exception raised then rest of the try block won't be executed even though we handled that exception hence we have to take only risky code inside try block and length of the try block should be as less as possible.

2).in addition to try block, there may be a chance of raising exception inside except and finally block also.

3). if any statement which is not part of try block raise an exception, then it is always abnormal termination.

❖ How to print exception information:

Example:

```
try:
   print(10/0)
except ZeroDivisionError as msg:
```

```
print("exception raised and its description is:",msg)
```

output:

```
exception raised and its description is: division by zero
```

❖ try with multiple except blocks:

- the way of handling exception is varied from exception to exception. Hence for every exception type a separate except block we have to provide. i.e. try with multiple except blocks is possible and recommended to use.

Example:

```
try:
    .....
    .....
    .....
except ZeroDivisionError:
    perform alternative arithmetic operations
except FileNotFoundError:
    use local file instead of remote file
```

- if try with multiple except blocks available then based on raised exception the corresponding except block will be executed.

Example:

```
try:
a=int(input("Enter 1st Number:"))
b=int(input("Enter 2nd Number:"))
print(a/b)
except ZeroDivisionError:
print("can't divide with zero")
except ValueError:
print("please provide int value only")
```

output:

```
Enter 1st Number:20
Enter 2nd Number:10
2.0
Enter 1st Number:30
Enter 2nd Number:0
can't divide with zero
Enter 1st Number:30
Enter 2nd Number:ten
please provide int value only
```

- if try with multiple except blocks available then the order of these except blocks is important. python interpreter will always consider from top to bottom until matched except block identified.

Example:

```
try:
    a=int(input("Enter 1st Number:"))
    b=int(input("Enter 2nd Number:"))
    print(a/b)
except ArithmeticError:
    print("ArithmeticErro Occurece")
```

```
except ValueError:
    print("please provide int value only")
except ZeroDivisionError:
    print("ZeroDivisionError Occurece")
```

output:

```
Enter 1st Number:30
Enter 2nd Number:0
ArithmeticErro Occurece
Enter 1st Number:10
Enter 2nd Number:50
0.2
Enter 1st Number:10
Enter 2nd Number:00
ArithmeticErro Occurece
```

❖ Single except block that can handle multiple exceptions:

- we can write a single except block that can handle multiple different types of exceptions.
- except (Exception1, Exception2, Exception3,...): OR
- except ((Exception1, Exception2,Exception3,...) as msg:
- parentheses are mandatory and this group of exceptions internally considered as tuple.

Example:

```
try:
    a=int(input("Enter 1st Number:"))
    b=int(input("Enter 2nd Number:"))
    print(a/b)
except ZeroDivisionError as msg:
    print("please provide valid numbers only and problem is:",msg)
```

output:

```
Enter 1st Number:30
Enter 2nd Number:3
10.0
Enter 1st Number:30
Enter 2nd Number:0
```

- please provide valid numbers only and problem is: division by zero

Example:

```
try:
    a=int(input("Enter 1st Number:"))
    b=int(input("Enter 2nd Number:"))
    print(a/b)
except (ZeroDivisionError,ValueError) as msg:
    print("please provide valid numbers only and problem is:",msg)
```

Output:

```
Enter 1st Number:30
Enter 2nd Number:six
please provide valid numbers only and problem is: invalid literal for int() with base 10: 'six'
Enter 1st Number:30
Enter 2nd Number:3
```

10.0
Enter 1st Number:30
Enter 2nd Number:0
please provide valid numbers only and problem is: division by zero

❖ Default except block:

- we can use default except block to handle any type of exceptions.
- in default except block generally we can print normal error messages.

syntax:
```
except:
  statements
```

Example:
```
try:
   a=int(input("Enter 1st Number:"))
   b=int(input("Enter 2nd Number:"))
   print(a/b)
except ZeroDivisionError:
   print("ZeroDivisionError:can't divide with zero")
except:
   print("Default Except: please provide valid input only")
```

output:
```
Enter 1st Number:20
Enter 2nd Number:0
ZeroDivisionError:can't divide with zero
Enter 1st Number:30
Enter 2nd Number:ten
Default Except: please provide valid input only
Enter 1st Number:30
Enter 2nd Number:3
10.0
```

Note: if try with multiple except blocks available then default except block should be last, otherwise we will get syntaxError.

Example:
```
try:
   print(30/0)
except:
   print("Default Except:")
except ZeroDivisionError:
   print("ZeroDivisionError")
```

output: SyntaxError: default 'except:' must be last

Note:

- the following are various possible combinations of except blocks
  1).except ZeroDivisionError:
  2).except ZeroDivisionError as msg:
  3).except (ZeroDivisionError, ValueError):
  4).except (ZeroDivisionError, ValueError) as msg:

❖ finally Block:

- it is not recommended to maintain clean up code(Resource Deallocating code or Resource releasing code) inside try block because there is no guarantee for the execution of every statement inside try block always.
- it is not recommended to maintain clean up code inside except block, because if there is no exception then except block won't be executed.
- Hence we required some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether exception handled or not handled. such type of best place is nothing but finally block.
- hence the main purpose of finally block is to maintain clean up code.

Example:

```
try:
   risky code
except:
   handling code
finally:
   clean-up code
```

- The speciality of finally block is it will be executed always whether exception raised or not raised and whether exception handled or not handled.

case-1: if there is no exception

Example:

```
try:
   print("try")
except:
   print("except")
finally:
   print("finally")
```

Output:

```
try
finally
```

case-2: if there is an exception raised but handled

Example:

```
try:
   print("try")
   print(10/0)
except ZeroDivisionError:
   print("except")
finally:
   print("finally")
```

output:

```
try
except
finally
```

case-3: if there is an exception raised but no handled

Example:

```
try:
```

```
    print("try")
    print(10/0)
except NameError:
    print("except")
finally:
    print("finally")
```

output:

```
try
finally
ZeroDivisionError: division by zero division by zero(abnormal termination
```

note: there is only one situation where finally block won't be executed ie whenever we are using os._exit(0) function.

- wehenever we are using os._exit(0) function then python virtual machine itself will be shutdown. in this particular case finally won't be executed.

Example:

```
imports
try:
    print("try")
    os._exit(0)
except NameError:
    print("except")
finally:
    print("finally")
```

output:

```
try
note:  os._exit(0)
```

- where 0 represents status code and it indicates normal termination there are multiple status codes are possible.

❖ Control flow in try-except-finally

```
try:
    statement-1
    statement-2
    statement-3
except:
    statement-4
finally:
    statement-5
    statement-6
```

case-1 if there is no exception
1,2,3,5,6 Normal termination

case-2 if an exception raised at statement-2 and the corresponding except block matched
1,4,5,6 Normal Termination

case-3 if an exception raised at statement-2 but the corresponding except block not matched 1,5 abnormal terminations

case-4 if an exception raised at statement-4 then it is always abnormal termination but before that finally block will be executed.

case-5 if an exception raised at statement-5 or at statement-6 then it is always abnormal termination

❖ Nested try-except-finally Blocks:

- we can take try-except-finally blocks inside try or except or finally blocks. i.e nesting of try-except-finally is possible.

```
try:
    .....
    .....
    try:
        .....
        .....
    except:
        .....
        .....
except:
    .....
    .....
```

- general risky code we have to take inside outer try block and too much risky code we have to take inside inner try block. Inside inner try block if an exception raised then inner except block is responsible to handle. if it is unable to handle then outer except block is responsible to handle.

Example:

```
try:
    print("outer try block")
    try:
        print("Inner try block")
        print(6/0)
    except ZeroDivisionError:
        print("Inner except block")
    finally:
        print("Inner finally block")
except:
    print("Outer except block")
finally:
    print("Outer finally block")
```

Output:
```
outer try block
Inner try block
Inner except block
Inner finally block
Outer finally block
```

❖control flow in nested try-except-finally:

```
try:
    statement-1
    statement-2
    statement-3
```

```
        try:
            statement-4
            statement-5
            statement-6
        except:
            statement-7
        finally:
            statement-8
            statement-9
    except y:
        statement-10
    finally:
        statement-11
        statement-12
```

case-1: if there is no exception
1,2,3,4,5,6,8,9,11,12 Normal Termination

case-2: if an exception raised at statement-2 and the corresponding except block matched
1,10,11,12 Normal Termination

case-3: if an exception raised at statement-2 and the corresponding except block not matched
1,11, Abnormal Termination

case-4: if an exception raised at statement-5 and inner except block matched
1,2,3,4,7,8,9,11,12 Normal termination

case-5: if an exception raised at statemet-5 and inner except block not matched but outer except block matched
1,2,3,4,8,11,12, Normal Termination

case-6: if an exception raised at statement-5 and inner and outer except blocks are not matched
1,2,3,4,8,12, Normal Termination

case-7: if an exception raised at statement-7 and the corresponding except block not matched
1,2,.,.,8,11, Abnormal Termination

case-8: if an exception raised at statement-5 and corresponding except block not matched
1,2,3,...,8,11,12 AbNormal termination

case-9: if an exception raised at statemet-8 and corresponding except block matched
1,2,3,4,8,11,12, Normal Termination

case-10: if an exception raised at statement-8 and corresponding except blocks not matched
1,2,..,.,.,11, abNormal Termination

case-11: if an exception raised at statement-9 and the corresponding except block matched
1,2,3.,.,8,10,11,12 normal Termination

case-12: if an exception raised at statement-9 and corresponding except block not matched
1,2,3,...,8,11, AbNormal termination

case-13: if an exception raised at statement-10 then it is always abnormal termination but before abnormal termination finally block(statement-11)will be executed.

case-14: if an exception raised at statement-11 or statement-12 then it is always abnormal termination.

Note: if the control entered into try block, then compulsory finally block will be executed.
if the control not entered into try block, then finally block won't be executed.

❖ else block with try-except-finally:

- we can use else block with try-except-finally blocks.
- else block will be executed if and only if there are no exceptions inside try block.

```
try:
   risky code
except:
   will be executed if exception inside try
else:
   well be executed if there is no exception inside try
finally:
   will be executed whether exception raised or not raised and handled or not handled
```

Example:

```
try:
   print("try")
   print(10/0)-->1
except:
   print("else")
finally:
   print("finally")
```

- if we comment line-1 then else block will be executed because there is no exception inside try. in this case the output is:

```
try
else
finally
```

- if we are not commenting line-1 then else block won't be executed because there is exception inside try block. in this case output is:

```
try
except   finally
```

various possible combinations of try-except-else-finally:

1) whenever we are writing try block, compulsory we should write except or finally block i.e without except of finally block we cannot write try block:
2) whenever we are writing except block, compulsory we should write try block i.e except without try is always invalid.
3) whenever we are writing finally block, compulsory we should write try block. i.e finally without try us always invalid.
4) we can write multiple except blocks for the same try, but we cannot write multiple finally blocks for the same try.
5) whenever we are writing else block compulsory except block should be there i.e without except we cannot write else block.
6) in try-except-else-finally order is important.
7) we can define try-except-else-finally inside try, except, else and finally blocks. i.e nesting of try-except-else-finally is always possible.

Types of exceptions:

- in python there are 2 types of exceptions are possible.
  1). predefined exceptions
  2).user defined exceptions

1)predefined exception:

- also known as inbuilt exception
- the exceptions which are raised automatically by python virtual machine whenever a particular even occurs are called pre-defined exceptions.

Example:

- whenever we are trying to perform division by zero, automatically python will raise ZeroDivisionError.
  print(20/0)

Example:

- whenever we are trying to convert input value to int type and if input value is not int value then python will raise valueError automatically
  x=int("ten")-->valueError

2).User defined Exceptions:

- also known as customized exceptions or programmatic exceptions
- some time we have to define and raise exceptions explicitly to indicate that something goes wrong, such type of exception is called user defined exceptions or customized exceptions
- programmer is responsible to define these exceptions and python not having any idea about these. hence, we have to rise explicitly based on our requirement by using "raise" keyword

Example:

InSuffientFundsException
InValidInputException
TooYoungException
TooOldException

❖How to define and raise customized exceptions:

- every exception in python is a class that extends exception class either directly or indirectly.

syntax:

- class classname(predefined exception class name):
  def __init__(self,arg):
  self.msg=arg

Example:

```
class TooYoungException(Exception):
    def __init__(self,arg):
        self.msg=arg
```

TooYoungException is our class name which is the child class of Exception
we can raise exception by using raise keyword as follows
raise TooYoungException("message")
example:

```
class TooYoungException(Exception):
    def __init__(self,arg):
```

```
        self.msg=arg
class TooYoungException(Exception):
    def __init__(self,arg):
        self.msg=arg
age=int(input("Enter Age:"))
if age>60:
    raise TooYoungException("Please wait some more time you will get best match soon!!!")
elif age<18:
    raise TooYoungException("your age already crossed marriage age...no chance of getting
marriage")
else:
    print("you will get match details soon by email!!!")
```

output:

```
Enter Age:27
you will get match details soon by email!!!
Enter Age:85
TooYoungException: Please wait some more time you will get best match soon!!!
Enter Age:9
TooYoungException: your age already crossed marriage age...no chance of getting marriage
```

MULTI THREADING

What is Multithreading?

- **Multithreading** is a programming technique that allows you to **run multiple tasks (threads) at the same time** within a single program. It's useful for tasks like:
- Downloading files
- Handling user input while doing background work
- Running animations or timers alongside main code

Why use Multithreading?

- To **perform multiple tasks at once**
- To make your program **faster and more responsive**
- Especially useful in **I/O-bound** tasks (e.g., reading files, waiting for user input)

How to Use Multithreading in Python?

- Python provides a built-in module called threading.

Basic Syntax:

```
import threading
def my_function():
    # code to be executed in a thread
    pass
t1 = threading.Thread(target=my_function)
t1.start()
```

Example 1: Printing messages in multiple threads

```
import threading
import time
def greet():
    for i in range(3):
        print("Hello from thread!")
        time.sleep(1)
# Create two threads
t1 = threading.Thread(target=greet)
t2 = threading.Thread(target=greet)
# Start threads
t1.start()
t2.start()
# Wait for both threads to finish
t1.join()
t2.join()
print("Main thread finished.")
```

Output:

```
Hello from thread!
Hello from thread!
Hello from thread!
Hello from thread!
Hello from thread!
Hello from thread!
Main thread finished.
```

Example 2: Threads with arguments

```
import threading
import time
def print_numbers(name):
    for i in range(1, 4):
        print(f"{name} prints {i}")
        time.sleep(1)
t1 = threading.Thread(target=print_numbers, args=("Thread A",))
t2 = threading.Thread(target=print_numbers, args=("Thread B",))
t1.start()
t2.start()
t1.join()
t2.join()
```

Example 3: Using Thread class with inheritance

```
import threading
import time
class MyThread(threading.Thread):
    def run(self):
        for i in range(3):
            print(f"Thread {self.name} running")
            time.sleep(1)
t1 = MyThread()
t2 = MyThread()
t1.start()
t2.start()
t1.join()
t2.join()
```

Note: Global Interpreter Lock (GIL) in Python

Python (CPython) has a limitation called the **Global Interpreter Lock (GIL)** which means:

- Only one thread runs Python bytecode at a time.
- So **CPU-bound** tasks (heavy computations) don't benefit much from multithreading.

For **CPU-bound** tasks, use **multiprocessing** instead of multithreading.

When to Use:

Task Type	Use
I/O Bound	Use Multithreading
CPU Bound	Use Multiprocessing

Real-World Uses:

- Downloading multiple files at once
- Chat applications
- Background timers
- UI responsiveness

❖ Multi-Tasking:

- Executing several tasks simultaneously is the concept of multitasking.
- ➢ There are 2 types of multi-tasking
  1). Process based multi-tasking
  2). Thread based multi-tasking

1. Process based multi-tasking:

- Executing several tasks simultaneously where each task is a separate independent process is called process based multi-tasking.

Example:

- While typing python program is the editor, we can listen mp3 audio song from the same system. At the same time, we can download a file from the internet. All these talks are executing simultaneously and independent of each other hence it is process based multi-tasking.
- This type of multi-tasking is suitable at operating system level.

2. Thread based multitasking:

- Executing several tasks simultaneously where each task is a separate independent part of the same program, is called thread based multi-tasking and each independent part is called a thread.
- This type of multi-tasking is best suitable at programmatic level.

Note: whether it is process based or thread based, the main advantage of multi-tasking is to improve performance of the system by reducing response time.

The main important application areas of multi-threading are:

1. To implement multimedia graphics
2. To develop animations
3. To develop video games
4. To develop web and application servers etc.

Note: where ever a group of independent jobs are available, then is highly recommended to execute simultaneously instead of executing one by one. For such type of cases, we should go for multi-threading.

- Python provides one inbuilt module “threading” to provide support for developing threads. Hence developing multi-threaded programs is very easy in python.
- Every python program by default contains one thread which is nothing but main thread.

Question: print the name of current executing thread

Program:

```
import threading
print(“Current Executing Thread:”, threading.current_thread().getName())
```

Output:

```
Current Executing Thread: Main Thread
```

Note: threading module contains function current thread () which returns current executing thread object. On this object if we call getName() method then we will get current executing thread name.

THREAD

- the ways of creating the thread in python
- we can create the thread in python by using 3 ways:
  1. creating a thread without using any class
  2.creating a thread by extending thread class
  3.creating a thread without extending thread class

1). creating a thread without using any class

Example:

```
from threading import*
def display():
   for i in range(1,4):
      print("child thread")
t=Thread(target=display)#Creating thread object
t.start()#starting of thread
for i in range(1,7):
   print("Main Thread")
```

Output:

```
child thread
child thread
child thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
```

- if multiple threads present to our program, then we cannot expect execution order and hence we cannot expect exact output for the multi-threaded programs because of this we cannot provide exact output for the above program. it is varied from machine to machine and run to run.
- note: thread is a pre-defined class present in threading module which can be used to create our own threads.

2) creating a thread by extending thread class

we have to create child class for thread class. in that child class we have to override run() method with our required job. whenever we call start() method then automatically run() method will be executed and performs our job.

Example:

```
from threading import*
class mythread(Thread):
   def run(self):
      for i in range(3):
         print("Child Thread-1")
r=mythread()
r.start()
for i in range(3):
```

```
print("Main Thread-1")
```

Output:

```
Child Thread-1
Child Thread-1
Child Thread-1
Main Thread-1
Main Thread-1
Main Thread-1
```

3) creating a Thread without extending Thread class

Example:

```
from threading import*
class test:
   def display(self):
      for i in range(6):
         print("child thread-2")
obj=test()
t=Thread(target=obj.display)
t.start()
for i in range(6):
   print("main thread-2")
```

output:

```
child thread-2
child thread-2
child thread-2
child thread-2
child thread-2
child thread-2
main thread-2
main thread-2
main thread-2
main thread-2
main thread-2
main thread-2
```

❖ without multi-threading

Example:

```
from threading import*
import time
def doubles(numbers):
   for n in numbers:
      time.sleep(1)
      print("Double:",2*n)
def squares(numbers):
   for n in numbers:
      time.sleep(1)
      print("square:",n*n)
numbers=[1,2,3,4,5,6]
begintime=time.time()
```

```
doubles(numbers)
doubles(numbers)
squares(numbers)
print("the total time take:", time.time()-begintime)
```

output:

```
Double: 2
Double: 4
Double: 6
Double: 8
Double: 10
Double: 12
square: 1
square: 4
square: 9
square: 16
square: 25
square: 36
the total time take: 12.055363655090332
```

❖ with multi-threading

Example:

```
from threading import*
import time
def doubles(numbers):
   for n in numbers:
      time.sleep(1)
      print("Double:",2*n)
def squares(numbers):
   for n in numbers:
      time.sleep(1)
      print("square:",n*n)
numbers=[1,2,3,4,5,6]
begintime=time.time()
t1=Thread(target=doubles,args=(numbers,))
t2=Thread(target=doubles,args=(numbers,))
t1.start()
t2.start()
t1.join()
t2.join()
# doubles(numbers)
# squares(numbers)
print("the total time take:", time.time()-begintime)
```

output:

```
Double:Double: 2
 2
Double:Double: 4
 4
Double:Double: 6
```

```
 6
Double:Double: 8
 8
Double:Double: 10
 10
Double:Double: 12
 12
the total time take: 6.058807134628296
```

❖ setting and getting name of a thread:

- Every thread in python has name. it may be default name generated by python or customized name provided by programmer.
- we can get and set name of thread by using the following thread class methods
- t.getName() --> Return Name of Thread
- t.setName(newName)-->To set our own name
- Note: every thread has implicit variable "name" to represent name of thread.

Example:

```
from threading import*
print(current_thread().getName())
current_thread().setName("Sangam Kumar")
print(current_thread().getName())
print(current_thread().name)
```

output:

```
Sangam Kumar
Sangam Kumar
Sangam Kumar
```

❖ thread identification number (ident):

- for every thread internally a unique identification number is available. we can access this id by using implicit variable "ident"

Example:

```
from threading import*
def test():
   print("child thread")
t=Thread(target=test)
t.start()
print("Main Thread identification Numbers:",current_thread().ident)
print("child Thread identification Number:",t.ident)
```

output:

```
child thread
Main Thread identification Numbers: 3748
child Thread identification Number: 8156
```

❖ active_count():

- this functions the number of active threads currently running.

Example:

```
from threading import*
import time
```

```
def display():
    print(current_thread().getName(),"...started")
    time.sleep(3)
    print(current_thread().getName(),"...ended")
print("the number of active threads:",active_count())
t1=Thread(target=display,name="childThread1")
t2=Thread(target=display,name="childThread2")
t3=Thread(target=display,name="childThread3")
t1.start()
t2.start()
t3.start()
print("The Number of active Threads:",active_count())
time.sleep(5)
print("The Number of active Threads:",active_count())
```

output:

```
the number of active threads: 6
childThread1 ...started
childThread2 ...started
childThread3 ...started
The Number of active Threads: 9
childThread2 ...ended
childThread1 ...ended
childThread3 ...ended
The Number of active Threads: 6
```

❖ enumerate() Function:

- this functions a list of all active threads currently running.

Example:

```
from threading import*
import time
def display():
    print(current_thread().getName(),"...started")
    time.sleep(3)
    print(current_thread().getName(),"...ended")
print("the number of active threads:",active_count())
t1=Thread(target=display,name="childThread1")
t2=Thread(target=display,name="childThread2")
t3=Thread(target=display,name="childThread3")
t1.start()
t2.start()
t3.start()
l=enumerate()
for t in l:
    print("Thread Name:",t.name)
time.sleep(5)
l=enumerate()
for t in l:
    print("Thread Name:",t.name)
```

output: the number of active threads: 6
childThread1 ...started
childThread2 ...started
childThread3 ...started
Thread Name: Sangam Kumar
Thread Name: IOPub
Thread Name: Heartbeat
Thread Name: Control
Thread Name: IPythonHistorySavingThread
Thread Name: Thread-4
Thread Name: childThread1
Thread Name: childThread2
Thread Name: childThread3
childThread1 ...ended
childThread3 ...ended
childThread2 ...ended
Thread Name: Sangam Kumar
Thread Name: IOPub
Thread Name: Heartbeat
Thread Name: Control
Thread Name: IPythonHistorySavingThread
Thread Name: Thread-4

❖ isAlive() Method:

- isAlive() Method checks whether a thread is still executing or not.

Example:

```
from threading import Thread, current_thread
import time
def display():
   print(current_thread().getName(), "...started")
   time.sleep(3)
   print(current_thread().getName(), "...ended")
t1 = Thread(target=display, name="childThread1")
t2 = Thread(target=display, name="childThread2")
t1.start()
t2.start()
print(t1.name, "is Alive:", t1.is_alive())  # Corrected function name
print(t2.name, "is Alive:", t2.is_alive())  # Corrected function name
time.sleep(5)
print(t1.name, "is Alive:", t1.is_alive())  # Corrected function name
print(t2.name, "is Alive:", t2.is_alive())  # Corrected function name
```

output:

childThread1 ...started
childThread2 ...started
childThread1 is Alive: True
childThread2 is Alive: True
childThread1childThread2 ...ended
...ended

```
childThread1 is Alive: False
childThread2 is Alive: False
```

❖ join() method:

- if a thread wants to wait until completing some other thread then we should go for join() method

Example:

```
from threading import Thread
import time
def display():
   for i in range(6):  # Corrected the typo here
      print("skills thread")
      time.sleep(2)
t = Thread(target=display)
t.start()
t.join()  # This line is executed by the main thread
for i in range(6):
   print("T3 Thread")
```

output:

```
skills thread
skills thread
skills thread
skills thread
skills thread
skills thread
T3 Thread
T3 Thread
T3 Thread
T3 Thread
T3 Thread
T3 Thread
```

Note: in the above example main thread waited until completing child thread. in this case note: we can call join() method with time period also.

t.join(seconds)

in this case thread will wait only specified amount of time.

Example:

```
from threading import Thread
import time
def display():
   for i in range(6):  # Corrected the typo here
      print("skills thread")
      time.sleep(2)
t = Thread(target=display)
t.start()
t.join(5)  # This line is executed by the main thread
for i in range(6):
   print("T3 Thread")
```

- in this case main thread waited only 5 seconds.

output:

```
skills thread
skills thread
skills thread
T3 Thread
T3 Thread
T3 Thread
T3 Thread
T3 Thread
T3 Thread
skills thread
skills thread
skills thread
```

summary of all methods related to threading module and thread

❖ Daemon Threads:

- the threads which are running in the background are called daemon threads
- the main objective of Daemon threads is to provide support for non-Daemon threads (like main thread)
- whenever main thread runs with low memory immediately PVM runs Garbage collector to destroy useless objects and to provide free memory, so that main thread can continue its execution without having any memory problems\
- we can check whether thread is Daemon or not by using t.isDaemon().method of thread class or by using daemon property.

Example:

```
from threading import*
print(current_thread().isDaemon())#False
print(current_thread().daemon)
```

output: False
False

- we can change Daemon nature by using setDaemon() method of Thread class.
- syntax: t.setDaemon(True)
- But we can use this method before starting of thread i.e once thread started, we cannot change its Daemon nature otherwise we will get
- RuntimeException:cannot set daemon status of active thread.

Example:

```
from threading import*
print(current_thread().isDaemon())#False
print(current_thread().setDaemon(True))
```

output:

- RuntimeError: cannot set daemon status of active thread

❖ Default Nature:

- By default main thread is always non-daemon but for the remaining threads daemon nature will be inherited from parent to child i.e if the parent thread is Daemon then child thread is also Daemon and if the parent thread is Non Daemon then childThread is also Non Daemon

Example:

```
from threading import*
def job():
   print("child Thread")
t=Thread(target=job)
print(t.isDaemon())
t.setDaemon(True)
print(t.isDaemon())
```

output:

```
False
True
```

Note: Main Thread is always non-Daemon and we cannot change its Daemon Nature Because it is already started at the beginning only

whenever the last non-Daemon thread terminates automatically all Daemon Threads will be terminated.

Example:

```
from threading import*
import time
def job():
   for i in range(6):
      print("Skills Thread")
      time.sleep(2)
t=Thread(target=job)
#t.setDaemon(True)===>line-1
t.start()
time.sleep(5)
print("End of the main thread")
```

output:

```
Skills Thread
Skills Thread
Skills Thread
End of the main thread
Skills Thread
Skills Thread
Skills Thread
```

❖ synchronization:

- if multiple threads are executing simultaneously then there may be a chance of data inconsistency problems

Example:

```
from threading import Thread
import time
def wish(name):
   for i in range(6):
      print("Good Evening: ", end="")
      time.sleep(2)
      print(name)


t1 = Thread(target=wish, args=("Sangam Kumar ",))  # Corrected the argument as a tuple
t2 = Thread(target=wish, args=("Satyam Kumar ",))  # Corrected the argument as a tuple
t1.start()
t2.start()
```

output:

```
Good Evening: Good Evening: Sangam Kumar
Good Evening: Satyam Kumar
Good Evening: Satyam Kumar Sangam Kumar
Good Evening:
Good Evening: Sangam Kumar
Good Evening: Satyam Kumar
Good Evening: Sangam Kumar Satyam Kumar
Good Evening:
Good Evening: Sangam Kumar Satyam Kumar
Good Evening:
```

Good Evening: Satyam Kumar Sangam Kumar

- we are getting irregular output because both threads are exciting simultaneously wish() function
- to overcome this problem, we should go for synchronization
- in synchronization the threads will be executed one by one so that we can overcome data inconsistency problems.
- synchronization means at a time only one thread

❖ the main application area of synchronization are

1. online Reservation system
2.Funds transfer from join accounts

- in python we can implement synchronization by using the following
  1). lock
  2). Rlock
  3).Semaphore

❖ synchronization by using lock concept

- Synchronization using locks is a way to ensure that multiple threads do not concurrently execute a critical section of code. In Python, you can use the threading module's Lock class to achieve this. Here's an example that demonstrates synchronization using locks:

Example:

```
import threading
# Define a global variable
counter = 0
# Create a lock
lock = threading.Lock()
# Function that increments the counter using the lock
def increment_counter():
    global counter
    for _ in range(1000000):
        # Acquire the lock
        lock.acquire()
        try:
            counter += 1
        finally:
            # Release the lock
            lock.release()
# Create two threads that increment the counter
thread1 = threading.Thread(target=increment_counter)
thread2 = threading.Thread(target=increment_counter)
# Start the threads
thread1.start()
thread2.start()
# Wait for both threads to finish
thread1.join()
thread2.join()
# Print the final value of the counter
print("Final Counter Value:", counter)
```

output:

Final Counter Value: 2000000

- In this example, we have a global variable counter, and two threads (thread1 and thread2) that increment the counter by 1 million times each. To ensure that the threads do not interfere with each other, we use a lock (lock) to synchronize access to the counter variable. When a thread wants to increment the counter, it first acquires the lock using lock.acquire(), then increments the counter, and finally releases the lock using lock.release().
- By using locks, we ensure that only one thread can access and modify the counter variable at a time, preventing race conditions and ensuring that the final counter value is correct.
- When you run this code, you should see that the final counter value is approximately 2 million, which is the expected result since both threads increment it by 1 million each.

problem with simple lock:

program:

```
from threading import*
l=Lock()
print("Main Thread trying to acquire Lock")
l.acquire()
print("Main Thread trying to acquire Lock Again")
l.acquire()
```

output: Main Thread trying to acquire Lock

Main Thread trying to acquire Lock Again

Program for synchronization by using RLock:

Program:

```
from threading import Thread, RLock
import time
l = RLock()
def factorial(n):
    l.acquire()
    if n == 0:
        result = 1
    else:
        result = n * factorial(n - 1)
    l.release()
    return result
def results(n):
    print("The Factorial of", n, "is:", factorial(n))
t1 = Thread(target=results, args=(6,))
t2 = Thread(target=results, args=(10,))
t1.start()
t2.start()
```

output:

The Factorial of 6 is: 720

The Factorial of 10 is: 3628800

#synchronization by using semaphore:

case-1: s=semaphore()

in this case se

important questions and answers

1. What is multithreading in Python?
Answer: Executing multiple threads (tasks) simultaneously to improve performance and responsiveness.
2. Which module is used for multithreading in Python?
Answer: threading module
3. What is a thread?
Answer: A lightweight subprocess used to perform tasks concurrently within a program.
4. How do you create a thread in Python?
Answer: Using threading.Thread(target=function_name).
5. How do you start a thread in Python?
Answer: By calling the .start() method on the thread object.
6. What does the .join() method do in multithreading?
Answer: It waits for the thread to complete before moving to the next line of code.
7. What is the Global Interpreter Lock (GIL)?
Answer: A mutex in CPython that allows only one thread to execute Python bytecode at a time.
8. What is a daemon thread?
Answer: A background thread that automatically exits when the main program ends.
9. How can you create a daemon thread in Python?
Answer: By setting daemon=True when creating the thread.
10. Which method pauses a thread temporarily?
Answer: time.sleep(seconds) pauses a thread for the given time.
Would you like me to prepare a short quiz or worksheet using these questions for practice or classroom use?
11. Can multiple threads run in true parallel in Python?
Answer: No, due to the GIL, only one thread executes Python code at a time (in CPython).
12. Which Python implementation allows true multithreading?
Answer: Jython and IronPython (they don't have GIL).
13. What is the difference between run() and start() in threads?
Answer: start() runs the thread asynchronously; run() runs it like a normal function.
14. How do you make a thread sleep in Python?
Answer: Use time.sleep(seconds).
15. How do you check if a thread is alive?
Answer: Use .is_alive() method.
16. What is the purpose of Lock in threading?
Answer: To prevent multiple threads from accessing shared data at the same time (avoid race conditions).
17. Which class is used to define custom threads in Python?
Answer: Subclass threading.Thread.
18. How can you stop a thread in Python safely?
Answer: Use a flag (boolean variable) to signal the thread to exit.
19. Can a thread be restarted in Python after it finishes?
Answer: No, a thread cannot be restarted once it is finished.
20. What are race conditions in multithreading?
Answer: A situation where multiple threads access shared data simultaneously leading to unpredictable behavior.

Mini Project

Project 1: File Downloader Simulation Using Multithreading

Description:

Simulates downloading multiple files at the same time using threads. Each file takes some time to "download" (simulated using sleep()), but multithreading lets us "download" all of them concurrently.

Code:

```
import threading
import time
def download_file(file_name):
    print(f"Starting download: {file_name}")
    time.sleep(2)  # Simulate time delay
    print(f"Finished downloading: {file_name}")
# List of files to "download"
files = ["video.mp4", "song.mp3", "document.pdf", "image.jpg", "presentation.pptx"]
threads = []
# Create and start a thread for each file
for file in files:
    t = threading.Thread(target=download_file, args=(file,))
    t.start()
    threads.append(t)
# Wait for all downloads to finish
for t in threads:
    t.join()
print("All files downloaded.")
```

Output:

```
Starting download: video.mp4
Starting download: song.mp3
Starting download: document.pdf
Starting download: image.jpg
Starting download: presentation.pptx
Finished downloading: song.mp3
Finished downloading: image.jpg
Finished downloading: presentation.pptx
Finished downloading: video.mp4
Finished downloading: document.pdf
All files downloaded.
```

Project 2: Real-time Clock + User Input Logger

Description:

One thread keeps printing the **current time** every second like a real-time clock, while another thread waits for **user input** and logs it.

Code:

```
import threading
import time
from datetime import datetime
# Thread 1: Display real-time clock
def show_clock():
    while True:
```

```
        now = datetime.now().strftime("%H:%M:%S")
        print(f"Time: {now}")
        time.sleep(1)

    # Thread 2: Take user input
    def log_user_input():
      while True:
        user_input = input("Type something (or 'exit' to quit): ")
        if user_input.lower() == "exit":
          print("Exiting input thread.")
          break
        print(f"You typed: {user_input}")

    # Create threads
    clock_thread = threading.Thread(target=show_clock, daemon=True)  # daemon so it auto-exits
    input_thread = threading.Thread(target=log_user_input)

    # Start threads
    clock_thread.start()
    input_thread.start()

    # Wait for user input thread to finish
    input_thread.join()
    print("Program Ended.")
```

Key Features:

- daemon=True lets the clock thread stop automatically when the main program ends.
- Runs **clock and input** simultaneously.

Project 3: Multithreaded Quiz Timer App

Description:

- One thread asks the user a question.
- Another thread runs a **countdown timer** (e.g., 10 seconds).
- If the user doesn't answer in time, the quiz ends automatically.

Code:

```
import threading
import time
# Global flag to check if user answered in time
answered = False
# Thread 1: Timer Countdown
def countdown_timer(seconds):
  global answered
  for i in range(seconds, 0, -1):
    if answered:
      return
    print(f"Time left: {i} seconds")
    time.sleep(1)
```

```
    if not answered:
        print("\nTime's up! You didn't answer in time.\n")
        exit()
# Thread 2: Ask the question
def ask_question():
    global answered
    question = "What is the capital of France?"
    correct_answer = "paris"
    answer = input(f"\n {question}\nYour answer: ")
    if answer.strip().lower() == correct_answer:
        print("Correct Answer!")
    else:
        print("Wrong Answer!")
    answered = True
# Create threads
timer_thread = threading.Thread(target=countdown_timer, args=(10,))
quiz_thread = threading.Thread(target=ask_question)

# Start both threads
timer_thread.start()
quiz_thread.start()

# Wait for user to finish
quiz_thread.join()
```

Sample Output (if answered in time):

```
        Time left: 10 seconds
        Time left: 9 seconds
        Time left: 8 seconds
What is the capital of France?
        Your answer: Paris
        Correct Answer!
```

Sample Output (if time runs out):

```
        Time left: 10 seconds
        Time left: 9 seconds
        Time left: 8 seconds
        ...
        Time's up! You didn't answer in time.
```

Features:

- Shows how multithreading can manage **simultaneous input and time constraints**.
- Can be extended into a full **quiz application** with scores, multiple questions, etc.

Would you like to extend this into a **complete multithreaded quiz app** with a scoreboard and multiple questions?

PYTHON DATABASE CONNECTION

- Python provides several libraries and modules for connecting to databases, allowing you to interact with various database management systems (DBMS) such as MySQL, PostgreSQL, SQLite, and more.

❖ SQLite Database Connection in Python:

- SQLite is a lightweight, serverless, self-contained, and transactional SQL database engine. To work with an SQLite database in Python, follow these steps:

1. Import the sqlite3 module: First, import the `sqlite3` module, which is part of the Python standard library.

Example:

```
import sqlite3
```

2. Connect to the SQLite database: To connect to an SQLite database, use the `connect()` method of the `sqlite3` module. You need to specify the name of the database file, and if it doesn't exist, SQLite will create it for you

```
# Create or connect to an SQLite database (e.g., 'mydatabase.db')
connection = sqlite3.connect('mydatabase.db')
```

3. Create a cursor object: A cursor allows you to interact with the database. You can execute SQL commands using this cursor.

```
# Create a cursor object
cursor = connection.cursor()
```

4. Execute SQL commands: You can now execute SQL commands such as creating tables, inserting data, querying data, updating data, or deleting data.

```
# Example: Creating a table
cursor.execute('''CREATE TABLE IF NOT EXISTS employees (
                    id INTEGER PRIMARY KEY,
                    first_name TEXT,
                    last_name TEXT,
                    email TEXT
                )''')
# Example: Inserting data
cursor.execute("INSERT INTO employees (first_name, last_name, email) VALUES (?, ?, ?)", ("John", "Doe", "john@example.com"))
# Example: Querying data
cursor.execute("SELECT * FROM employees")
data = cursor.fetchall()
for row in data:
    print(row)
# Example: Updating data
cursor.execute("UPDATE employees SET email = ? WHERE id = ?", ("new_email@example.com", 1))
# Example: Deleting data
cursor.execute("DELETE FROM employees WHERE id = ?", (1,))
```

5. Commit and close: After executing the necessary database operations, make sure to commit the changes and close the connection.

```
# Commit the changes
```

```
connection.commit()
# Close the cursor and the connection
cursor.close()
connection.close()
Here's a complete example:
import sqlite3
# Connect to or create an SQLite database
connection = sqlite3.connect('mydatabase.db')
cursor = connection.cursor()
# Create a table
cursor.execute('''CREATE TABLE IF NOT EXISTS employees (
                id INTEGER PRIMARY KEY,
                first_name TEXT,
                last_name TEXT,
                email TEXT
            )''')
# Insert data
cursor.execute("INSERT INTO employees (first_name, last_name, email) VALUES (?, ?, ?)", ("John",
"Doe", "john@example.com"))
# Query data
cursor.execute("SELECT * FROM employees")
data = cursor.fetchall()
for row in data:
   print(row)
# Update data
cursor.execute("UPDATE employees SET email = ? WHERE id = ?", ("new_email@example.com",
1))
# Delete data
cursor.execute("DELETE FROM employees WHERE id = ?", (1,))
# Commit changes and close the connection
connection.commit()
cursor.close()
connection.close()
```

❖ mysql Database Connection in Python:

- To connect to a MySQL database in Python, you can use the `mysql-connector-python` library, which provides an easy way to interact with MySQL databases.
- Below are the step-by-step instructions along with an example of connecting to a MySQL database and performing various operations.

Step 1: Install the `mysql-connector-python` Library

- You need to install the `mysql-connector-python` library if it's not already installed. You can do this using `pip`:
  - bash
  - pip install mysql-connector-python

Step 2: Import the Required Modules

Import the necessary modules from the `mysql.connector` package:

- python
- import mysql.connector

Step 3: Establish a Connection

- To establish a connection to your MySQL database, you need to provide connection parameters such as host, user, password, and database name. Replace these values with your specific database configuration:
- python

- Replace these values with your MySQL server configuration

```
host = "your_host"
user = "your_username"
password = "your_password"
database = "your_database_name"
# Establish a connection
connection = mysql.connector.connect(
  host=host,
  user=user,
  password=password,
  database=database
)
```

Step 4: Create a Cursor

- Create a cursor object to interact with the database:
- python

```
cursor = connection.cursor()
```

Step 5: Execute SQL Queries

- You can execute SQL queries using the cursor object. Here are some examples:
  - Create a Table
  - python
  - Example: Creating a table

```
create_table_query = """
CREATE TABLE IF NOT EXISTS employees (
  id INT AUTO_INCREMENT PRIMARY KEY,
```

```
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    email VARCHAR(255)
)
"""
cursor.execute(create_table_query)
```

- Insert Data

- python
- Example: Inserting data

```
insert_query = """
INSERT INTO employees (first_name, last_name, email)
VALUES (%s, %s, %s)
"""
data_to_insert = ("Sangam", "Kumar", "sangam@example.com")
cursor.execute(insert_query, data_to_insert)
```

 # Commit the changes

```
connection.commit()
```

- Query Data
- python
- Example: Querying data

```
select_query = "SELECT * FROM employees"
cursor.execute(select_query)
result = cursor.fetchall()
for row in result:
  print(row)
```

- Update Data
- python
- Example: Updating data

```
update_query = "UPDATE employees SET email = %s WHERE id = %s"
new_email = ("new_email@example.com", 1)
cursor.execute(update_query, new_email)
# Commit the changes
connection.commit()
```

- Delete Data
- Example: Deleting data

```
delete_query = "DELETE FROM employees WHERE id = %s"
employee_id_to_delete = (1,)
cursor.execute(delete_query, employee_id_to_delete)
# Commit the changes
connection.commit()
```

Step 6: Close the Cursor and Connection

- After you have finished working with the database, close the cursor and the connection:

```
python
cursor.close()
connection.close()
```

❖ Complete Example:

```
import mysql.connector

# Replace these values with your MySQL server configuration
host = "your_host"
user = "your_username"
password = "your_password"
database = "your_database_name"
# Establish a connection
connection = mysql.connector.connect(
    host=host,
    user=user,
    password=password,
    database=database
)

# Create a cursor
cursor = connection.cursor()
# Create a table
create_table_query = """
CREATE TABLE IF NOT EXISTS employees (
    id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    email VARCHAR(255)
)
cursor.execute(create_table_query)
# Insert data
insert_query = """
INSERT INTO employees (first_name, last_name, email)
VALUES (%s, %s, %s)
"""
data_to_insert = ("John", "Doe", "john@example.com")
cursor.execute(insert_query, data_to_insert)
# Query data
select_query = "SELECT * FROM employees"
cursor.execute(select_query)
result = cursor.fetchall()
for row in result:
    print(row)
# Update data
update_query = "UPDATE employees SET email = %s WHERE id = %s"
new_email = ("new_email@example.com", 1)
cursor.execute(update_query, new_email)
# Delete data
delete_query = "DELETE FROM employees WHERE id = %s"
employee_id_to_delete = (1,)
```

```
cursor.execute(delete_query, employee_id_to_delete)
# Commit the changes
connection.commit()
# Close the cursor and the connection
cursor.close()
connection.close()
```

❖ MongoDB database connection in Python:

- To connect to a MongoDB database in Python, you can use the `pymongo` library, which provides an interface for working with MongoDB.
- Below are the step-by-step instructions along with an example of connecting to a MongoDB database and performing various operations.

Step 1: Install the `pymongo` Library

- If you don't have the `pymongo` library installed, you can install it using `pip`:
- bash

- pip install pymongo

Step 2: Import the Required Modules

- Import the necessary modules from the `pymongo` package:
- python

- import pymongo

Step 3: Establish a Connection

- To establish a connection to your MongoDB database, you need to provide connection parameters such as host and port. Replace these values with your specific MongoDB server configuration:

```
# Replace these values with your MongoDB server configuration
host = "your_host"
port = 27017  # Default MongoDB port
# Establish a connection
client = pymongo.MongoClient(host, port)
```

Step 4: Access a Database

- You can access a specific database within your MongoDB server using the client object. If the database doesn't exist, MongoDB will create it for you when you first access it:

```
# Access a database (replace 'your_database_name' with your actual database name)
db = client.your_database_name
```

Step 5: Access a Collection

- In MongoDB, data is organized into collections, which are similar to tables in relational databases. You can access a collection within the database:

```
# Access a collection (replace 'your_collection_name' with your actual collection name)
collection = db.your_collection_name
```

Step 6: Perform CRUD Operations

- You can perform various CRUD (Create, Read, Update, Delete) operations on the collection. Here are some examples:

- Insert Documents

Example: Insert a document

```
document_to_insert = {
  "first_name": "John",
  "last_name": "Doe",
  "email": "john@example.com"
}
insert_result = collection.insert_one(document_to_insert)
print(f"Inserted document ID: {insert_result.inserted_id}")
```

- Find Documents

Example: Find documents

```
find_query = {"first_name": "John"}
results = collection.find(find_query)
for result in results:
   print(result)
```

- Update Documents

Example: Update documents

```
update_query = {"first_name": "John"}
new_data = {"$set": {"email": "new_email@example.com"}}
update_result = collection.update_many(update_query, new_data)
print(f"Modified {update_result.modified_count} documents")
```

- Delete Documents

Example: Delete documents

```
delete_query = {"first_name": "John"}
delete_result = collection.delete_many(delete_query)
print(f"Deleted {delete_result.deleted_count} documents")
```

Step 7: Close the Connection

➢ After you have finished working with the database, close the connection:

```
client.close()
```

❖ Complete Example:

```
import pymongo
# Replace these values with your MongoDB server configuration
host = "your_host"
port = 27017  # Default MongoDB port
# Establish a connection
client = pymongo.MongoClient(host, port)
# Access a database
db = client.your_database_name
# Access a collection
collection = db.your_collection_name
# Insert a document
document_to_insert = {
   "first_name": "sangam",
   "last_name": “kumar",
   "email": "sangam@example.com"
}
insert_result = collection.insert_one(document_to_insert)
print(f"Inserted document ID: {insert_result.inserted_id}")
# Find documents
find_query = {"first_name": "John"}
results = collection.find(find_query)
for result in results:
   print(result)
# Update documents
update_query = {"first_name": "John"}
new_data = {"$set": {"email": "new_email@example.com"}}
```

```
update_result = collection.update_many(update_query, new_data)
print(f"Modified {update_result.modified_count} documents")
# Delete documents
delete_query = {"first_name": "John"}
delete_result = collection.delete_many(delete_query)
print(f"Deleted {delete_result.deleted_count} documents")
# Close the connection
client.close()
```

❖ MongoDB Atlas database in Python:

Example:

```
import pymongo
# Replace <connection_string> with your actual MongoDB Atlas connection string
connection_string = "<connection_string>"
# Establish a connection to MongoDB Atlas
client = pymongo.MongoClient(connection_string)
# Access a database
db = client.your_database_name
# Access a collection
collection = db.your_collection_name
# Insert a document
document_to_insert = {
   "first_name": "Sangam",
   "last_name": "Kumar",
   "email": "sangam@example.com"
}
insert_result = collection.insert_one(document_to_insert)
print(f"Inserted document ID: {insert_result.inserted_id}")
# Find documents
find_query = {"first_name": "John"}
results = collection.find(find_query)
for result in results:
   print(result)
# Update documents
update_query = {"first_name": "John"}
new_data = {"$set": {"email": "new_email@example.com"}}
update_result = collection.update_many(update_query, new_data)
print(f"Modified {update_result.modified_count} documents")
# Delete documents
delete_query = {"first_name": "John"}
delete_result = collection.delete_many(delete_query)
print(f"Deleted {delete_result.deleted_count} documents")
# Close the connection
client.close()
```

API in python

What is an API?

API stands for **Application Programming Interface**.

It allows two software applications to **talk to each other**.

Example:

- Google Maps API lets developers show maps in their apps.
- Weather API lets you get weather data from a website into your Python app.

Types of APIs:

1. **Web APIs** (most common):
   - Communicate over the internet using **HTTP/HTTPS**.
   - Example: REST APIs, GraphQL APIs
2. **Library/Module APIs**:
   - Functions inside Python libraries like math, os, etc.
3. **Hardware APIs**:
   - APIs that interact with hardware (e.g., camera, sensors).
4. **Operating System APIs**:
   - Functions provided by OS (e.g., Windows API for file access).

How to Create a Simple API in Python?

We'll use the **Flask** framework (easy and popular for beginners).

Step 1: Install Flask

```
pip install flask
```

Step 2: Simple API Code

```
from flask import Flask, jsonify, request
app = Flask(__name__)
# Sample data
students = [
  {"id": 1, "name": "Tanmay", "marks": 88},
  {"id": 2, "name": "Sangam", "marks": 95}
]
# GET: Fetch all students
@app.route('/students', methods=['GET'])
def get_students():
  return jsonify(students)
# POST: Add a new student
@app.route('/students', methods=['POST'])
def add_student():
  data = request.get_json()
  students.append(data)
  return jsonify({"message": "Student added!", "student": data}), 201
# Run the API
if __name__ == '__main__':
  app.run(debug=True)
```

Output:

Visit: http://127.0.0.1:5000/students

You'll see the list of students in JSON format.

API Methods (HTTP Verbs):

Method	Purpose	Example
GET	Read data	Get all users
POST	Create new data	Add a new user
PUT	Update existing data	Edit a user's details
DELETE	Delete data	Remove a user

Summary:

- **API** connects software.
- **Web API** is the most common.
- Use **Flask** to build simple APIs in Python.
- Use **GET, POST, PUT, DELETE** to interact with data.

Would you like a project using this Flask API with a database or frontend (like React or HTML)?

RESTful API in Python

- Creating a RESTful API in Python involves several steps, including setting up a web framework, defining routes and endpoints, handling requests and responses, and implementing the API logic.
- In this example, we'll create a simple RESTful API using the Flask framework, which is a lightweight and popular choice for building APIs in Python.

What is a RESTful API?

REST stands for **Representational State Transfer**.

A **RESTful API** is a type of **web API** that follows certain rules so it can be easily used to **Create, Read, Update, and Delete (CRUD)** data over the internet using HTTP methods like:

- GET → Read data
- POST → Create data
- PUT → Update data
- DELETE → Delete data

Real-life Example:

Imagine a **notebook app** that stores notes.

A RESTful API for it might have:

URL	HTTP Method	Action
/notes	GET	Get all notes
/notes/1	GET	Get note #1
/notes	POST	Add new note
/notes/1	PUT	Update note #1
/notes/1	DELETE	Delete note #1

How to Build a RESTful API in Python (using Flask)

Step 1: Install Flask pip install flask

Step 2: Sample REST API Code

from flask import Flask, jsonify, request

```
app = Flask(__name__)
# Sample data (like a database)
notes = [
    {"id": 1, "title": "Buy milk"},
    {"id": 2, "title": "Study Python"}
]
# GET all notes
@app.route('/notes', methods=['GET'])
def get_notes():
    return jsonify(notes)
# GET a single note by ID
@app.route('/notes/<int:note_id>', methods=['GET'])
def get_note(note_id):
    for note in notes:
        if note["id"] == note_id:
            return jsonify(note)
    return jsonify({"error": "Note not found"}), 404
# POST a new note
@app.route('/notes', methods=['POST'])
def create_note():
    new_note = request.get_json()
    notes.append(new_note)
    return jsonify(new_note), 201
# PUT update a note
@app.route('/notes/<int:note_id>', methods=['PUT'])
def update_note(note_id):
    for note in notes:
        if note["id"] == note_id:
            note["title"] = request.get_json()["title"]
            return jsonify(note)
    return jsonify({"error": "Note not found"}), 404
# DELETE a note
@app.route('/notes/<int:note_id>', methods=['DELETE'])
def delete_note(note_id):
    for note in notes:
        if note["id"] == note_id:
            notes.remove(note)
            return jsonify({"message": "Note deleted"})
    return jsonify({"error": "Note not found"}), 404
# Run the API
if __name__ == '__main__':
    app.run(debug=True)
```

How to Test?

- Use **Postman** or **curl**
- Or just open a browser for GET requests:
  - http://127.0.0.1:5000/notes

Step 1: Install Flask

- First, make sure you have Flask installed. If you don't, you can install it using `pip`:
- bash
- pip install Flask

Step 2: Create a Flask Application

- Create a Python file (e.g., `app.py`) to build your API. Import Flask and initialize the application:

```
from flask import Flask
app = Flask(__name__)
```

Step 3: Define Routes and Endpoints

- Define the routes and endpoints for your API. Routes specify the URLs that your API will respond to, and endpoints are specific functions that handle those URLs. Here's an example with a single endpoint that returns a JSON response:

```
@app.route('/')
def hello_world():
  return {'message': 'Hello, World!'}
```

Step 4: Run the Application

- To run your Flask application, add the following code at the end of your script:

```
if __name__ == '__main__':
  app.run(debug=True)
```

Step 5: Start the API

- Run your Flask application using the following command in your terminal:
- `bash
- python app.py
- Your API should be accessible at `http://127.0.0.1:5000/` by default. Accessing this URL in your web browser or using a tool like `curl` should return the "Hello, World!" message in JSON format.

Step 6: Test Additional Endpoints

- You can define additional endpoints by creating new routes and functions in your Flask application. Here's an example of an endpoint that returns a list of items:

```
@app.route('/items', methods=['GET'])
def get_items():
  items = [{'id': 1, 'name': 'Item 1'}, {'id': 2, 'name': 'Item 2'}]
  return {'items': items}
```

Step 7: Accept and Respond to Parameters

- You can accept parameters in your API requests and respond accordingly. Here's an example of an endpoint that accepts a parameter in the URL and returns data based on that parameter:

```
@app.route('/item/<int:item_id>', methods=['GET'])
def get_item(item_id):
  # Assuming you have a data source or database to fetch data from
  item = find_item_by_id(item_id)
  if item:
    return {'item': item}
  else:
    return {'message': 'Item not found'}, 404
```

Step 8: Handle Request Data

- You can also handle data sent in the request body. Here's an example of an endpoint that accepts JSON data in a POST request and returns a response:

```
from flask import request
@app.route('/add_item', methods=['POST'])
def add_item():
    data = request.json  # Assuming JSON data is sent in the request body
    # Process the data and add the item to your data source or database
    return {'message': 'Item added successfully'}
```

Step 9: Error Handling

- Handle errors and exceptions gracefully in your API. Flask allows you to return specific HTTP status codes and error messages in your responses, as shown in previous examples.

Step 10: Deploy Your API

- To make your API accessible on the internet, you'll need to deploy it on a web server. Popular options for deployment include platforms like Heroku, AWS, Azure, and more. The deployment process will vary depending on your chosen platform.

❖ Complete Example for ReastFull API creating in python:

- Creating a complete RESTful API in Python involves several components, including setting up a web framework, defining routes, handling HTTP requests, and implementing the API logic.
- In this example, we'll create a RESTful API using the Flask framework with comments to explain each step. We'll create a simple "to-do list" API with endpoints to create, read, update, and delete tasks.

Step 1: Install Flask

- Make sure you have Flask installed. If not, install it using `pip`:
- bash
- pip install Flask

Step 2: Create a Flask Application

- Create a Python file (e.g., `app.py`) and start building your API.
- from flask import Flask, request, jsonify

```
app = Flask(__name__)
```

Step 3: Define a Data Structure

- For this example, we'll use a Python list to store tasks. In a real application, you might use a database.
- tasks = []

Step 4: Define Routes and Endpoints

- Define the routes and endpoints for your API. We'll create routes for listing tasks, adding tasks, getting a single task, updating a task, and deleting a task. Comments explain each endpoint.

Endpoint to create a new task (POST request)

```
@app.route('/tasks', methods=['POST'])
def create_task():
    data = request.get_json()
    if 'title' in data:
        task = {
            'id': len(tasks) + 1,
            'title': data['title'],
            'done': False
        }
        tasks.append(task)
```

```
        return jsonify({'message': 'Task created successfully'}), 201
    else:
        return jsonify({'message': 'Title is required'}), 400
# Endpoint to list all tasks (GET request)
@app.route('/tasks', methods=['GET'])
def get_tasks():
    return jsonify({'tasks': tasks})
# Endpoint to get a single task by ID (GET request)
@app.route('/tasks/<int:task_id>', methods=['GET'])
def get_task(task_id):
    task = next((task for task in tasks if task['id'] == task_id), None)
    if task:
        return jsonify({'task': task})
    else:
        return jsonify({'message': 'Task not found'}), 404
# Endpoint to update a task by ID (PUT request)
@app.route('/tasks/<int:task_id>', methods=['PUT'])
def update_task(task_id):
    data = request.get_json()
    task = next((task for task in tasks if task['id'] == task_id), None)
    if task:
        task['title'] = data.get('title', task['title'])
        task['done'] = data.get('done', task['done'])
        return jsonify({'message': 'Task updated successfully'})
    else:
        return jsonify({'message': 'Task not found'}), 404
# Endpoint to delete a task by ID (DELETE request)
@app.route('/tasks/<int:task_id>', methods=['DELETE'])
def delete_task(task_id):
    task = next((task for task in tasks if task['id'] == task_id), None)
    if task:
        tasks.remove(task)
        return jsonify({'message': 'Task deleted successfully'})
    else:
        return jsonify({'message': 'Task not found'}), 404
```

Step 5: Run the Application

Add the following code at the end of your script to run the Flask application.

```
if __name__ == '__main__':
    app.run(debug=True)
```

Step 6: Start the API

- Run your Flask application using the following command in your terminal:
- bash
- python app.py

Your API should be accessible at `http://127.0.0.1:5000/` by default.

Step 7: Test the API

- You can use tools like `curl` or Postman to test your API endpoints. Here are some example API requests:

- Create a task:
  bash
  curl -X POST -H "Content-Type: application/json" -d '{"title": "Task 1"}' http://127.0.0.1:5000/tasks
- List all tasks:
  `bash
  curl http://127.0.0.1:5000/tasks
- Get a single task by ID:
  bash
  curl http://127.0.0.1:5000/tasks/1
  ```
- Update a task by ID:

  curl -X PUT -H "Content-Type: application/json" -d '{"title": "Updated Task 1", "done": true}' http://127.0.0.1:5000/tasks/1
- Delete a task by ID:
  ```bash
  curl -X DELETE http://127.0.0.1:5000/tasks/1

# 50 advanced Python interview questions and answers:

1. What is the Global Interpreter Lock (GIL) in Python?
   - The Global Interpreter Lock (GIL) is a mutex that allows only one thread to execute in a Python process at a time, preventing true multi-threaded parallel execution of Python code.
2. Explain the use of decorators in Python.
   - Decorators are functions that modify or enhance other functions or methods. They are often used for tasks like logging, authentication, and performance profiling.
3. What is the purpose of generators in Python?
   - Generators are a way to create iterators in Python. They allow you to iterate over a potentially large sequence of items without storing them all in memory simultaneously.
4. How do you handle exceptions in Python?
   - You can use `try`, `except`, and `finally` blocks to handle exceptions. The `try` block contains the code that might raise an exception, the `except` block handles the exception, and the `finally` block is for optional cleanup.
5. Explain the use of the `collections` module in Python.
   - The `collections` module provides specialized container datatypes like `namedtuple`, `Counter`, and `deque` that are alternatives to the built-in types.
6. What are metaclasses in Python?
   - Metaclasses are classes that define the behavior of other classes (class factories). They allow you to customize class creation and behavior.
7. What is the purpose of the `__init__` method in Python classes?
   - `__init__` is a special method (constructor) that is automatically called when an object of a class is created. It initializes the object's attributes.
8. Explain the difference between shallow copy and deep copy in Python.
   - A shallow copy creates a new object but inserts references to the original object's elements. A deep copy creates a new object and recursively copies all elements, including nested objects.
9. What is monkey patching in Python?
   - Monkey patching is a technique where you modify or extend classes or modules at runtime to change or add functionality.
10. How do you create a virtual environment in Python?
    - You can create a virtual environment using the `venv` module in Python: `python -m venv myenv`.
11. What is a closure in Python?
    - A closure is a nested function that remembers and has access to variables in the containing (enclosing) function's local scope, even after the outer function has finished executing.
12. Explain the purpose of the `asyncio` library in Python.
    - `asyncio` is a library for asynchronous programming in Python. It provides tools for writing concurrent code using the `async` and `await` keywords.
13. How does garbage collection work in Python?
    - Python uses reference counting and a cyclic garbage collector to reclaim memory. Objects are deallocated when they are no longer referenced.
14. What is the Global Namespace in Python?
    - The Global Namespace refers to the top-level namespace in Python, where global variables and functions are defined. It's the outermost scope accessible throughout the program.
15. What is the difference between a module and a package in Python?
    - A module is a single Python file that contains functions, classes, and variables. A package is a directory that contains multiple modules and a special `__init__.py` file to indicate it's a package.

16. How can you profile Python code for performance optimization?
- You can use tools like `cProfile`, `line_profiler`, and `memory_profiler` to profile Python code and identify performance bottlenecks.
17. What is the purpose of the `itertools` module in Python?
- The `itertools` module provides a collection of fast, memory-efficient tools for working with iterators. It includes functions for creating iterators for permutations, combinations, and more.
18. Explain the Global Interpreter Lock (GIL) and its impact on multi-threaded Python programs.
- The GIL is a mutex that allows only one thread to execute Python bytecode at a time. This can limit the performance benefits of multi-threading in CPU-bound tasks, as only one thread can execute Python code simultaneously. However, it can still provide benefits in I/O-bound tasks.
19. What is the purpose of the `contextlib` module in Python?
- The `contextlib` module provides utilities for working with context managers, such as the `with` statement. It simplifies resource management and cleanup.
20. How can you create a custom exception in Python?
- You can create a custom exception by defining a new class that inherits from the `Exception` base class.
21. Explain the difference between a shallow copy and a deep copy in Python with examples.
- A shallow copy creates a new object but inserts references to the original object's elements. A deep copy creates a new object and recursively copies all elements, including nested objects. Here's an example:

```
import copy

original_list = [[1, 2, 3], [4, 5, 6]]
# Shallow copy
shallow_copy = copy.copy(original_list)
# Deep copy
deep_copy = copy.deepcopy(original_list)
```

22. What is method resolution order (MRO) in Python, and how is it determined in multiple inheritance?
- MRO defines the order in which base classes are searched when a method is called on an object. In multiple inheritance, Python uses the C3 Linearization algorithm to determine the MRO.
23. How does Python's garbage collection system work, and what are cyclic references?
- Python uses reference counting and a cyclic garbage collector. Cyclic references occur when objects reference each other in a cycle, making them unreachable through normal reference counting. The cyclic garbage collector identifies and collects such objects
24. Explain the purpose of the `functools` module in Python.
- The `functools` module provides higher-order functions and operations on callable objects. It includes functions like `partial`, `reduce`, and `lru_cache`.
25. What is the Global Interpreter Lock (GIL) in Python, and how does it affect multi-threading?
- The GIL is a mutex that allows only one thread to execute Python bytecode at a time. This can limit the performance benefits of multi-threading for CPU-bound tasks. However, it may still provide benefits for I/O-bound tasks due to Python's efficient I/O operations.
26. Explain the purpose of the `__slots__` attribute in Python classes.
- The `__slots__` attribute allows you to explicitly define the attributes that a class can have. It can optimize memory usage and prevent the creation of additional instance attributes.
27. What is the Global Interpreter Lock (GIL) in Python, and how does it impact multi-core processors?

- The GIL is a mutex that allows only one thread to execute Python bytecode at a time. It can impact multi-core processors because it prevents Python from fully utilizing multiple CPU cores for CPU-bound tasks. However, it still allows for concurrency in I/O-bound tasks.

28. How do you implement a Singleton design pattern in Python?**
- A Singleton ensures that a class has only one instance. You can implement it using a class variable to store the instance and a static method to access or create the instance.

29. Explain the use of the `@staticmethod` decorator in Python.
- The `@staticmethod` decorator is used to define static methods in a class. Static methods are not bound to an instance and can be called on the class itself.

30. What is the Global Interpreter Lock (GIL) in Python, and how does it impact multi-threaded programs?
- The GIL is a mutex that allows only one thread to execute Python bytecode at a time. It can limit the parallel execution of Python code in multi-threaded programs, especially for CPU-bound tasks. However, it doesn't prevent multi-threading in I/O-bound programs.

31. How can you implement a stack in Python using a list?
- You can use a list to implement a stack in Python by using the `append()` method to push elements onto the stack and the `pop()` method to remove elements from the top of the stack.

32. Explain the purpose of the `collections.namedtuple` function in Python.
- `collections.namedtuple` is a factory function that creates a new class for creating simple, immutable data structures. It's useful for creating lightweight objects with named fields.

33. What is the purpose of the `__str__` and `__repr__` methods in Python classes?
- `__str__` returns a human-readable string representation of an object, while `__repr__` returns an unambiguous string representation for developers.

34. How do you implement a custom context manager in Python using the `with` statement?
- You can create a custom context manager by defining a class with `__enter__` and `__exit__` methods. The `with` statement manages the context and calls these methods.

35. Explain the purpose of the `enum` module in Python.
- The `enum` module provides a way to create and work with enumerated constants. It helps improve code readability and maintainability by giving meaningful names to values.

36. What is the purpose of the `functools.partial` function in Python?
- `functools.partial` is a function that allows you to fix a certain number of arguments of a function and generate a new function with those fixed arguments.

37. How do you handle circular imports in Python?
- Circular imports occur when two or more modules import each other. You can resolve them by using techniques like importing inside functions or using the `import` statement where needed.

38. Explain the purpose of the `threading` module in Python.
- The `threading` module provides a high-level interface for creating and managing threads in Python. It simplifies multi-threading and concurrent programming.

39. What is the Global Interpreter Lock (GIL) in Python, and why does it exist?
- The GIL is a mutex that allows only one thread to execute Python bytecode at a time. It exists to simplify memory management in CPython (the standard Python implementation) and ensure thread safety.

40. How can you implement a custom iterator in Python using the `__iter__` and `__next__` methods?
- To create a custom iterator, you define a class with `__iter__` and `__next__` methods. `__iter__` returns the iterator object itself, and `__next__` returns the next value in the sequence.

41. Explain the purpose of the `__getitem__` and `__setitem__` methods in Python classes.
- `__getitem__` allows you to access an object's elements using square brackets like `obj[key]`, and `__setitem__` allows you to assign values to those elements.
42. How can you create a Python package and use it in your code?
- To create a Python package, create a directory with an `__init__.py` file and place your module files inside it. You can then import and use the package in your code.
43. Explain the purpose of the `itertools.chain` function in Python.
- `itertools.chain` is used to combine multiple iterable objects into a single iterator. It's often used to concatenate lists or other iterable sequences efficiently.
44. What is the purpose of the `contextlib.contextmanager` decorator in Python?
- `contextlib.contextmanager` is used to create context managers as generator functions. It simplifies the creation of custom context managers using the `with` statement.
45. How do you implement a custom metaclass in Python, and what is its use case?
- You can implement a custom metaclass by defining a class that inherits from `type`. Metaclasses are used to customize class creation and behavior, often for frameworks and libraries.
46. Explain the use of the `multiprocessing` module in Python.
- The `multiprocessing` module allows you to create and manage multiple processes, enabling parallel execution of code on multi-core processors.
47. What is the purpose of the `collections.Counter` class in Python?
- `collections.Counter` is a dictionary subclass that counts the occurrences of elements in a collection. It's often used for tallying items in a sequence.
48. How do you implement a custom iterable in Python?
- To create a custom iterable, define a class with an `__iter__` method that returns an iterator object. The iterator object should implement a `__next__` method to produce values.
49. Explain the use of the `async` and `await` keywords in Python for asynchronous programming.
- `async` is used to define asynchronous functions, and `await` is used within asynchronous functions to pause execution until an awaited coroutine is complete. Together, they enable asynchronous programming.
50. What is the Global Interpreter Lock (GIL) in Python, and how does it affect multi-threaded performance?
- The GIL is a mutex that allows only one thread to execute Python bytecode at a time. This can limit the performance benefits of multi-threading for CPU-bound tasks. However, it may still provide benefits for I/O-bound tasks due to Python's efficient I/O operations.

What is RID Organization (RID संस्था क्या

- **RID Organization** यानि **Research, Innovation and Discovery Organization एक** संस्था हैं जो TWF (TWKSAA WELFARE FOUNDATION) **NGO** द्वारा RUN किया जाता है | जिसका मुख्य उदेश्य हैं आने वाले समय में सबसे पहले **NEW (RID, PMS & TLR)** की खोज, प्रकाशन एवं उपयोग भारत की इस पावन धरती से भारतीय संस्कृति, सभ्यता एवं भाषा में ही हो |
- देश, समाज, एवं लोगों की समस्याओं का समाधान **NEW (RID, PMS & TLR)** के माध्यम से किया जाये इसके लिए ही **इस RID Organization** की स्थपना 30.09.2023 किया गया है | जो TWF द्वारा संचालित किया जाता है |
- TWF (TWKSAA WELFARE FOUNDATION) NGO की स्थपना 26-10-2020 में बिहार की पावन धरती सासाराम में Er. RAJESH PRASAD एवं Er. SUNIL KUMAR द्वारा किया गया था जो की भारत सरकार द्वारा मान्यता प्राप्त संस्था हैं |
- Research, Innovation & Discovery में रूचि रखने वाले आप सभी विधार्थियों**,** शिक्षकों एवं बुधीजिवियो से मैं आवाहनं करता हूँ की आप सभी **इस RID संस्था** से जुड़ें एवं अपने बुध्दि, विवेक एवं प्रतिभा से दुनियां को कुछ नई **(RID, PMS & TLR)** की खोजकर, बनाकर एवं अपनाकर लोगों की समस्याओं का समाधान करें |

MISSION, VISSION & MOTIVE OF "RID ORGANIZATION"	
मिशन	हर एक ONE भारत के संग
विजन	TALENT WORLD KA SHRESHTM AB AAYEGA भारत में और भारत का TALENT भारत में
मकसद	NEW (RID, PMS, TLR)

MOTIVE OF RID ORGANIZATION NEW (RID, PMS, TLR)		
NEW (RID)		
R	I	D
Research	Innovation	Discovery
NEW (TLR)		
T	L	R
Technology, Theory, Technique	Law	Rule
NEW (PMS)		
P	M	S
Product, Project, Production	Machine	Service

RID रीड संस्था की मिशन, विजन एवं मकसद को सार्थक हमें बनाना हैं |
भारत के वर्चस्व को हर कोने में फैलना हैं |
कर के नया कार्य एक बदलाव समाज में लाना हैं |
रीड संस्था की कार्य-सिध्दांतों से ही, हमें अपनी पहचान बनाना हैं |

Er. Rajesh Prasad (B.E, M.E)
Founder:
TWF & RID Organization

Advance Python के इस **E-Book** में अगर मिलती त्रुटी मिलती है तो कृपया हमें सूचित करें | **WhatsApp's No: 9202707903 or**
Email Id: ridorg.in@gmail.com

।। धन्यवाद ।।