T3 SKILLS CENTER
T3 कौशल केंद्र

Foundation Day
30-09-2023

TWKSAA ADVANCE PYTHON

Er. Rajesh Prasad

- आप एक सागर हो बहते नदी का जल नहीं आप एक बदलाव हो भटकाव की कोई राह नहीं
- उस रास्ते पर चलो जिस रास्ते पर भीड़ कम हो (हर काम हो कुछ अलग)
- देश की मिट्टी से करो आप इतना प्यार जहाँ जाओ वहाँ मिले खूब इज्जत और सम्मान
- छह दिन कीजिए अपना काम एक दिन कीजिए त्वक्सा को दान
- त्वक्सा एक चिंगारी हैं हर जगह जलना हम सब की जिमेवारी हैं
- **Motive: - New (RID PMS & TLR)**

"त्वक्सा एडवांस पायथन के इस पुस्तक में आप एडवांस पायथन के संबंध में सभी बुनियादी अवधारणाएँ सीखेंगे। मुझे आशा है कि इस 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 TWKSAA Advance python book you will learn all basic concept regarding Advance 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".

"Skill कौशल एक व्यक्ति के पास उनके ज्ञान, अनुभव, तत्वशास्त्रीय योग्यता, और प्रैक्टिकल अभियांत्रिकी के साथ संचित नौकरी, व्यापार, या अन्य चुनौतीपूर्ण परिस्थितियों में सक्रिय रूप से काम करने की क्षमता को कहते हैं। यह व्यक्ति के द्वारा सीखी जाने वाली कौशलों की प्रतिभा, क्षमता और निपुणता को संक्षेप में व्यक्त करता है"।

TWKSAA RID MISSION

(Research)	(Innovation)	(Discovery)
अनुसंधान करने के महत्वपूर्ण कारण:	**नवीनीकरण करने के महत्वपूर्ण कारण:**	**खोज करने के महत्वपूर्ण कारण:**
1. नई ज्ञान की प्राप्ति	1. प्रगति के लिए	1. नए ज्ञान की प्राप्ति
2. समस्याओं का समाधान	2. परिवर्तन के लिए	2. ज्ञान के विकास में योगदान
3. तकनीकी और व्यापार में उन्नति	3. उत्पादन में सुधार	3. अविष्कारों की खोज
4. विकास को बढ़ावा देना	4. प्रतिस्पर्धा में अग्रणी होने के लिए	4. समस्याओं का समाधान
5. सामाजिक प्रगति	5. समाज को लाभ	5. समाज के उन्नति का माध्यम
6. देश विज्ञान और प्रौद्योगिकी का विकास	6. देश विज्ञान और प्रौद्योगिकी के विकास।	6. देश विज्ञान और तकनीक के विकास

"T3 Skills Center is a Learning Earning and Development Based Skill Center."

T3 कौशल केंद्र एक सीखने कमाई और विकास आधारित कौशल केंद्र है।

T3 SKILLS CENTER

OOPS

- OOPs, which stands for Object-Oriented Programming, is a programming paradigm that organizes data and behavior into reusable structures called objects. It is a way of designing and structuring code to represent real-world entities and their interactions.

❖ 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 the 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)
```

➢ 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)

➢ 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

Example:

```
class student:
  ''' Developed by twksaa skills center for python demo'''
  def __init__(self):
    self.name='skills'
    self.age=30
    self.marks=99
  def talk(self):
    print("Hello i am:", self.name)
    print("My Age is:", self.age)
    print("My marks are:", self.marks)
```

Example:

```
        class A:#creation of class
            #class body
            x=30#class variable
            def __init__(s,a,b):#a,b----instance variables
                s.a=a
                s.b=b
            def dis(s):
                print(s.a,s.b)#accessing instance variables
                print(s.x)#accessing class variable can be done by
                print(A.x)#self reference as well as class name
                #print(A.a,A.b) instance variables can't be accessed by class names
        ob = A(10,20)
        print(type(ob))
#Accessing class variables
        A.x#by using the class name
        ob.x#by using the object/instance
        #Accessing instance variables
        ob.a
        ob.b
        #A.a   instance variables can't be accessed by class names
        #ob.dis()
        ob1=A(20,30)
        ob2=A(30,40)
#same value of class variables across all objects provided it's not modified
        # print(ob1.x)
        # print(ob2.x)
        # print(ob.x)
        # print(A.x)
#instance variables
        print(ob.a)
        print(ob.b)
        print(ob1.a)
        print(ob1.b)
        print(ob2.a)
        print(ob2.b)
# 1. class variables can be accessed by class name,self reference & object/instance
# 2. instance variables can only be accessed by object/instance,self reference not by class name
#instance vs class vs static methods
        class A:
            x=30
            def __init__(y,a,b):
                y.a=a
                y.b=b
            def dis(s):#instance method-- it should have self reference
                print(s.a,s.b)#instance method can access both class & instance variables
                print(s.x)
```

```
   #both class method & static method are used to access class variables only
   @classmethod
   def clsmethod(cls):#class method needs a class reference
      print(cls.x)
   @staticmethod
   def stmethod():#static method needs no arguments
      print(A.x)
ob = A(10,20)
ob.clsmethod()
#ob.dis()
#ob.stmethod()
class Cat:
   species="mammal"
   def __init__(self,name,age):
      self.name=name
      self.age=age
#1---instantiate the cat object with 3 cats
cat1=Cat("Jimmy",25)
cat2=Cat("Tommy",30)
cat3=Cat("Meesha",22)
#2---find the age of the oldest cat using user-defined function
def find_oldest_cat(*args):
   return max(args)
#3--print the age of oldest cat using the function in 2(use fstring)
print(f"Age of the oldest cat is-{find_oldest_cat(cat1.age,cat2.age,cat3.age)}")
```

❖ What is Object:

- Physical existence of a class is noting but object. We can create any number of objects for a class

Syntax:

Reference variable = class name()

Example:

S=skills()

❖ 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.

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 is:", self.name)
```

```
        print("My Rollno is:", self.rollno)
        print("My marks are:", self.marks)
    obj1=student("sangam", 39,100)
    obj1.skills()
```

Output:

```
Hello my is: sangam
My Rollno is: 39
My marks are: 100
```

❖ Self-Variable:

- Self is the default variable which is always pointing to current object (like this keyword in java)
- By using self, we can access instance variables and instance methods of object.

Note:

1) Self should be first parameter inside constructor
   Example:
   ```
   def __init__(self):
   ```
2) Self should be first parameters inside instance methods
   Example:
   ```
   def skills (self):
   ```

❖ Constructor:

- Constructor is a special method in python.
- The name of the constructor should be __init__(self)
- Constructor will be executed automatically at the time of object creation.
- The main purpose of constructor is to declare and initialize instance variables.
- Per object constructor will be executed only once
- Constructor can take at least one argument (at least self)
- Constructor is optional and if we are not providing any constructor then python will provide default constructor.

Example:

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

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

❖ types of variables:

- there are three types of variables are allowed.
  1. instance variables (object level variables)
  2.static variables (class level variables)
  3. local variables (method level 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
```

❖ Types of methods:

- there are three types of methods
  1.instance method
  2.class method
  3.static method

1). instance methods:

- inside method implementation if we are using instance variables then such types of method are called instance methods.
- inside instance method declaration, we have to pass self-variable. def m1(self):
- but using self-variable inside method we can able to access instance variables.
- within the class we can call instance method by using self-variable and from outside of the class we can call by using object reference.

example:

```
class student:
  def _init_(self,name,marks):
    self.name=name
    self.marks=marks
  def display(self):
    print('Hey',self.name)
    print('your marks are:',self.marks)
  def grade(self):
    if self.marks>=60:
      print('you got first Grade')
    elif self.marks>=50:
      print('you got second Grade')
    elif self.marks>=35:
      print('you got third Grade')
    else:
      print('Fail')
n=int(input("Enter the Number of student:"))
for i in range(n):
  name=input("Enter name")
  marks=int(input('Enter Marks:'))
  s=student(name,marks)
  s.display()
  s.grade()
  print()
```

output:

```
Enter the Number of students:3
Enter name Sangam Kumar
Enter Marks:100
Hey Sangam Kumar
your marks are: 100
you got first Grade

Enter name Satyam Kumar
Enter Marks:99
Hey Satyam Kumar
your marks are: 99
you got first Grade

Enter name Raushni Kumari
Enter Marks:98
Hey Raushni Kumari
your marks are: 98
you got first Grade
```

❖ setter and getter methods:

- setter can set and get the values of instance variables by using getter and setter methods.
- setter method:
- setter methods can be used to set values to the instance variables. setter methods also known as mutator methods.

syntax:

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

example:

```
def setVaraible(self.name):
        self.name=name
```

❖ Getter Method:

- Getter methods can be used to get values of the instance variables. Getter methods also known as accessor methods.

syntax:

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

Example:

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

Enter The Name: Sujeet Kumar
Enter the marks: 89
Hi Sujeet Kumar
Your Marks are: 89

Enter The Name: Sangam Kumar
Enter the marks: 99
Hi Sangam Kumar
Your Marks are: 99
```

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
```

❖ **<u>inner classes:</u>**

- 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:

- in old language like c++, programming is responsible for both creation and destruction of objects usually programmer taking very much care while creating object, but neglecting destruction of useless objects, because of his reflectance, total memory can be filled with useless object which create memory problem and total application will be down with out of memory error.
- but in python we have some assistant which is always running in the background to destroy useless objects because this assistant the chance of failing python program with memory problems is very less. this assistant is nothing but Garbage collector.
- hence the main objective of garbage collector is to destroy useless objects.
- if an object does not have any reference variable, then that object eligible for garbage collection.

❖ How to enable and disable Garbage collector in our program:

- by default, Garbage collector is enabled, but we can disable based on our requirement. in this context we can use the following functions of gc module.
    1. gc.isenabled() --> Return True if GC enabled
    2.gc.disable()--> To disable GC explicitly
    3. gc.enable()-->To enable GC explicitly

Example:

```
import gc
print(gc.isenabled())
gc.disable()
print(gc.isenabled())
gc.enable()
print(gc.isenabled())
```

output:

```
True
False
True
```

❖ Destructions:

- Destruction is a special method and the name should be __del__
- just before destroying an object Garbage collector always calls destructor to perform clean-up activities (Resource deallocation activities like close database connection etc.)
- once destruction execution completed then Garbage collector automatically destroys that object.
- Note: the job of destructor is not to destroy object and it is just to perform clean-up activities.

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
```

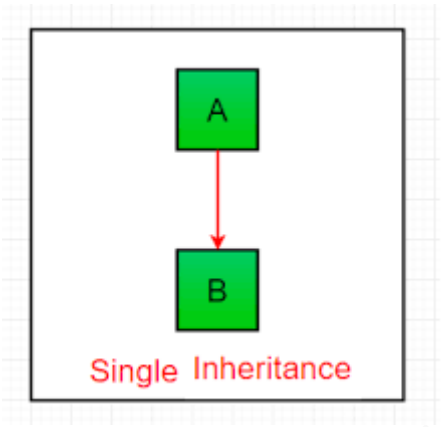
❖ What is inheritance ?:

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

1-Single Inheritance



example:

```
class A:#Base class/ parent class
   def __init__(s,a,b):
      s.a=a
      s.b=b
   def disp(self):
      print(self.a)
class B(A):#Child class
    def disp1(self):
      print("Number", self.a,self.b)
obj=B(10,20)
obj.disp()
obj.disp1()
```

output:

```
10
Number 10 20
```

➢ By using super class:

```
class A:#Base class/ parent class
   def __init__(s,a,b):
      s.a=a
      s.b=b
```

```
    def disp(self):
        print(self.a,self.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}")

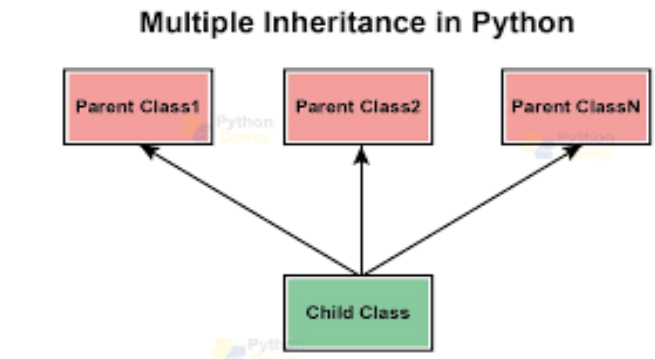
ob=B(10,20,30)
ob.dis()
```

output:

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

❖ Multiple Inheritance:

Diagram:



Example:

```
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}")
```

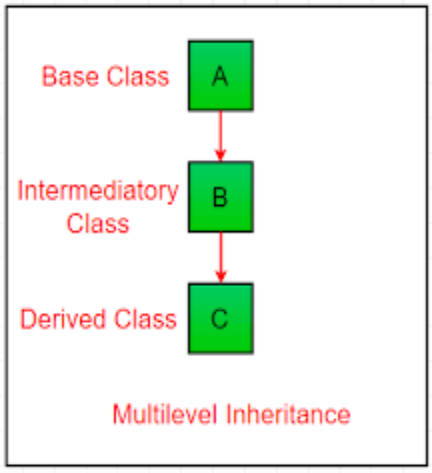
```
ob=Child("Charlie","Ms Charlie","John")
ob.dis()
Child.mro()
```

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]
```

❖ Multi-level Inheritance:

Diagram:



Example:

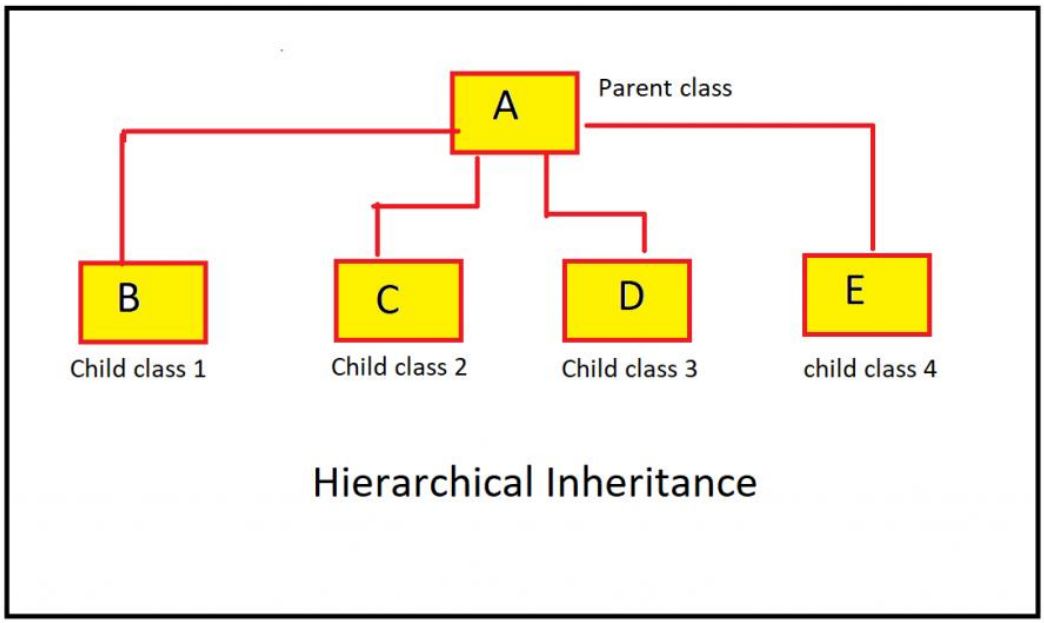
```
class GrandFather:#Parent class
    def __init__(s,son):
        s.son=son
class Father(GrandFather):#Intermediate
    def __init__(s,son,grandson):
        super().__init__(son)
        s.grandson=grandson
class Son(Father):#Child class
    def __init__(s,son,grandson,child):
        super().__init__(son,grandson)
        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
```

❖ **Hierarchical Inheritance:**

Diagram:



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()
```

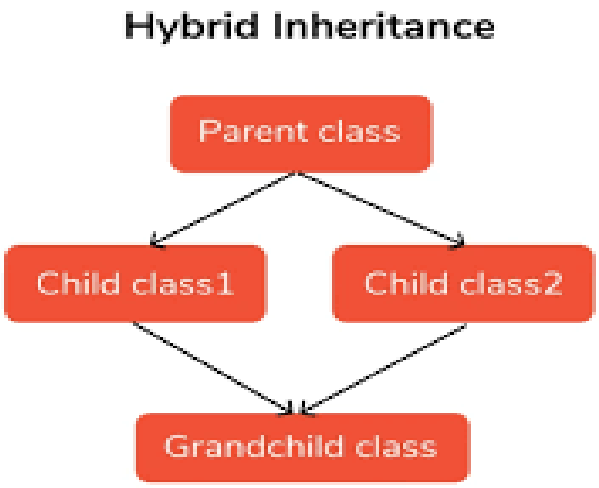
output:

```
[<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
```

❖ 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
```

❖ Method Resolution order (MRO)

- in hybrid inheritance the method resolution order is decided based on MRO algorithm.
- this algorithm is also known as C3 algorithm.
- semuele pedroni proposed this algorithm.
- if follows DLR (depth left to right)
- left parent will get more priority than right parent.
- MRO(x)=x+merge (MRO(p1),MRO(p2),....parentList)
- Head elemet vs Tail Terminology:
- assume c1,c2,c3.....are class.
- in the list: c1c2c3c4c5....
- c1 is considered as head element and remaining is considered as tail.

❖ How to find merge:

- take the head of first list
- if the list head is not in the tail part of any other list then add this head to the result and remove it from the list in the merge.
- if the head is present in the tail part of any other list, then consider the head element of the next list and continue the same process.

Note: we can find MRO of any class by using mro()function.

```
print(classname.mro())
```

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']
```

Example: for method resolution order:

diagram:

mro(a)=a,object
mro(b)=b,a,object
mro(c)=c,a,object
mro(d)=d,b,c,a,object

program:

```
class a:
   pass
class b(a):
   pass
class c(a):
   pass
class d(b,c):
   pass
print(a.mro())
print(b.mro())
print(c.mro())
print(d.mro())
```

output:

```
[<class '__main__.a'>, <class 'object'>]
[<class '__main__.b'>, <class '__main__.a'>, <class 'object'>]
[<class '__main__.c'>, <class '__main__.a'>, <class 'object'>]
[<class '__main__.d'>, <class '__main__.b'>, <class '__main__.c'>, <class
'__main__.a'>, <class 'object'>
```

Example-2

daigram

mro(a)=a,object
mro(b)=b,object
mro(c)=c,object

mro(x)=x,a,b,object
mro(y)=y,b,c,object
mro(p)=p,x,a,y,b,c, object

program:

```
class a:
   pass
class b:
   pass
class c:
   pass
class x(a, b):
   pass
class y(b,c):
   pass
class p(x,y,c):
   pass
print(a.mro())#ao
print(x.mro())#xabo
print(y.mro())#ybco
print(p.mro())#pxaybco
```

output:

```
[<class '__main__.a'>, <class 'object'>]
[<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

diagram:

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)
```

program:

```
class d:
    pass
class e:
    pass
class f:
    pass
class b(d,e):
    pass
class c(d,f):
    pass
class a(b,c):
    pass
```

```
print(d.mro())
print(b.mro())
print(c.mro())
print(a.mro())
```

output:

```
[<class '__main__.d'>, <class 'object'>]
[<class '__main__.b'>, <class '__main__.d'>, <class '__main__.e'>, <class 'object'>]
[<class '__main__.c'>, <class '__main__.d'>, <class '__main__.f'>, <class 'object'>]
[<class '__main__.a'>, <class '__main__.b'>, <class '__main__.c'>, <class '__main__.d'>,
<class '__main__.e'>, <class '__main__.f'>, <class 'object'>]
```

❖ super () method:

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

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-2:

```
class p:
   a=30
   def __init__(s):
      s.b=9
   def m1(s):
      print("parent instance method")
   @classmethod
```

```
    def m2(skills):
        print("parent class method")
    @staticmethod
    def m3():
        print("parent static method")
class c(p):
    a=393
    def __init__(s):
        s.b=333
        super().__init__()
        print(super().a)
        super().m1()
        super().m2()
        super().m3()
r=c()
```

output:

```
30
parent instance method
parent class method
parent static method
```

- in the above example we are using super() to call various members of parent class.
- here calll method of a particular super class:
- we can use the following approaches

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
```

❖ important points about super():

case-1

- from child class we are not allowed to access parent class instance variables by using super(), compulsory we should use self only
- but we can access parent class static variables by using super().

example:

```
class p:
  a=30
  def __init__(s):
    s.b=50
class c(p):
  def m1(s):
    print(super().a)#valid
    print(s.b)#valid
    #print(super().b)#invalid attributeError 'super' object has no attribute 'b'
r=c()
r.m1()
```

output

```
30
50
```

case-2 :

- from child class constructor and instance method, we can access parent class instance method static method and class method by using super()

Example:

```
class p:
  def __init__(s):
    print("parent constructor")
  def m1(s):
    print("parent instance method")
  @classmethod
  def m2(skills):
    print("parent class method")
  @staticmethod
  def m3():
    print("parent static method")
class c(p):
  def __init__(s):
    super().__init__()
    super().m1()
    super().m2()
```

```
        super().m3()
    def m1(s):
        super().__init__()
        super().m1()
        super().m2()
        super().m3()
r=c()
r.m1()
```

output:

```
parent constructor
parent instance method
parent class method
parent static method
parent constructor
parent instance method
parent class method
parent static method
```

case-3 :

- from child class class method we cannot access parent class instance methods and constructors by using super() directly (but indirectly possible). but we can access parent class static and class methods.

example:

```
class p:
    def __init__(s):
        print("parent constructor")
    def m1(s):
        print("parent instance method")
    @classmethod
    def m2(skills):
        print("parent class method")
    @staticmethod
    def m3():
        print("parent static method")
class c(p):
    @classmethod
    def m1(skills):
        #super().__init__()-->invalid
        #super().m1()--->invalid
        super().m2()
        super().m3()
c.m1()
```

output:

```
parent class method
parent static method
```

- from class method of child class. how to call parent class instance methods and constructors:

example:

```
class a:
```

```
    def __init__(s):
        print("parent constructor")
    def m1(s):
        print("parent instance method")
class b(a):
    @classmethod
    def m2(skills):
        super(b,skills).__init__(skills)
        super(b,skills).m1(skills)
b.m2()
```

output: parent constructor
parent instance method

case-4 :

- in child class static method we are not allowed to use super() generally (but in special way we can use)

example:

```
class p:
    def __init__(self):
        print("parent constructor")
    def m1(self):
        print("parent instance method")
    @classmethod
    def m2(skills):
        print("parent class method")
    @staticmethod
    def m3():
        print("parent static method")
class c(p):
    @staticmethod
    def m1():
        #super().m1()-->invalid
        #super().m2()-->invalid
        #super().m3()--invalid
c.m1()
```

RuntimeError: super():no arguments

❖ **How to call parent class static method from child class static method by using super():**

example:

```
class A:
    @staticmethod
    def m1():
        print("parent static method")
class B(A):
    @staticmethod
    def m2():
        super(B,B).m1()
B.m2()
```

output: parent static method

POLYMORPHISM

- poly means many, morph means forms.
- polymorphism means 'many forms'
- 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

example2: + and * operator acts concatenation arithmetic addition, multiplication and repetition operator

Example-3: the same method with different implementations in parent class and child classes. (overriding)

❖ **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. Method overriding-** It is achieved using Inheritance

```
class A:#PArent class
   def dis(s):
      print("class A")
class B(A):#child class
   def dis(s):
      print("Class B")
ob=B()
ob.dis()
```

output:

```
Class B
```

❖ **2. Method Overloading-** In Python, it can be implemented by default arguments

Example:

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

```
        s.d=d
    def dis(s):
        print(f"The values are:\n{s.a}\n{s.b}\n{s.c}\n{s.d}")
ob1=A(10)#1 arg
ob1.dis()
ob2=A(10,20)#2 args
ob2.dis()
ob3=A(10,50,60)#3 args
ob3.dis()
ob4=A(1,2,3,4)#4 args
ob4.dis()
#In this example, init is called 4 times with different number of args
```

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
```

1) .Duck typing philosophy of python:

- in python we cannot specify the type explicitly. based on provided value at runtime the type will be considered automatically. hence python is considered as dynamically typed programming language.

```
def f1(obj):
obj.talk()
```

➢ what is the type of object?

- we cannot decide at the beginning.at runtime we can pass any type. then how we can decide the type?
- at runtime if it walks like a duck and talks like a duck, it must be duck python follows this principle. this is called duck typing philosophy of python.

example:

```
class duck:
    def talk(s):
```

```
        print('Quack... Quack...')
    class dog:
      def talk(s):
        print("Bow Bow...")
    class cat:
      def talk(s):
        print("Moew Moew...")
    class goat:
      def talk(s):
        print("Myaah Myaah...")
    def f1(obj):
      obj.talk()
    l=[duck(),dog(),goat()]
    for obj in l:
      f1(obj)
```

output:

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

- the problem in this approach is if obj does not contain talk() method then we will get attributeError.

example:

```
class duck:
  def talk(s):
    print('Quack... Quack...')
class dog:
  def bark(s):
    print("Bow Bow...")
def f1(obj):
  obj.talk()
d=duck()
f1(d)
d=dog()
f1(d) #AttributeError: 'dog' object has no attribute 'talk'
```

output:

```
Quack... Quack...
AttributeError: 'dog' object has no attribute 'talk'
```

but we can solve this problem by using hasattr() function.

hasattr(obj),'attributename')--> attributename can be method name or variable name

example:

```
class duck:
  def talk(s):
    print("Quack... Quack...")
class human:
  def talk(s):
    print("hello hi kaun...")
def f1(obj):
```

```
    if hasattr(obj,'talk'):
        obj.talk()
    elif hasattr(obj,"bark"):
        obj.bark()
d=duck()
f1(d)
h=human()
f1(h)
d=dog()
f1(d)
```

output:

```
Quack... Quack...
hello hi kaun...
Bow Bow...
```

2). Overloading:

- we can use same operator or method for different purpose.

example-1:

+operator can be used for arithmetic addition and string concatenation

```
print(3+6)#9
print('skills'+'center')#skillscenter
```

example-2:

* operator can be used for multiplication and string repetition purposes

```
print(3*6)#18
print('skills'*3)#skillsskillsskills
```

example-3: we can deposited() method to deposited cash or cheque or dd

```
deposit(cash)
deposit(cheque)
deposit(dd)
```

❖ **there are three types of overloading**

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

1). operator overloading:

- we can use the same operator for multiple purpose, which is nothing but operator overloading
- python support operator overloading

example-1: +operator can be used for arithmetic addition and string concatenation

```
print(3+6)#9
print('skills'+'center')#skillscenter
```

example-2: * operator can be used for multiplication and string repetition purposes

```
print(3*6)#18
print('skills'*3)#skillsskillsskills
```

example:

```
class book:
    def __init__(s,pages):
        s.pages=pages
b1=book(100)
```

```
b2=book(200)
print("the total number of pages=",b1+b2)
TypeError: unsupported operand type(s) for +: 'book' and 'book'
```

- we can overload + operator to work with book object also. i.e python supports operator overloading
- for every operator magic method are available. to overload any operator we have to override that method is our class.
- internally + operator is implemented by using __add__() method. this method is called magic method for +operator.we have to override this method in our class.
- program to overload + operator for our book class objects.

Example:

```
class book:
   def __init__(s,pages):
      s.pages=pages
   def __add__(s,other):
      return s.pages+other.pages
b1=book(100)
b2=book(200)
print("the total number of pages=",b1+b2)
```

output:

```
the total number of pages= 300
```

the following is the list of operators and corresponding magic methods.

1.+ --> object.__add__(self,other)
2.- --> object.__sub__(self,other)
3.* --> object.__mul__(self,other)
4./ --> object.__div__(self,other)
5.// --> object.__floordiv__(self,other)
6.% --> object.__mod__(self,other)
7.** --> object.__pow__(self,other)
8.+= --> object.__iadd__(self,other)
9.-= --> object.__isub__(self,other)
10.*= --> object.__imul__(self,other)
11./= --> object.__idiv__(self,other)
12.//= --> object.__ifloordiv__(self,other)
13.%= --> object.__imod__(self,other)
14.**= --> object.__ipow__(self,other)
15.< --> object.__lt__(self,other)
16.<= --> object.__le__(self,other)
17.> --> object.__gt__(self,other)
18.>= --> object.__ge__(self,other)
19.== --> object.__eq__(self,other)
20.!= --> object.__ne__(self,other)

- **overloading >and <= operators for student class objects:**

Example:

```
class student:
   def __init__(s,name,marks):
```

```
        s.name=name
        s.marks=marks
    def __gt__(s,other):
        return s.marks>other.marks
    def __le__(self,o):
        return self.marks<=o.marks
print("10>20=",10>20)
s1=student("sangam",100)
s2=student("satyam",99)
print("s1>s2",s1>s2)
print("s1<s2",s1<s2)
print("s1>=s2",s1>=s2)
print("s1<=s2",s1<=s2)
```

output:

```
10>20= False
s1>s2 True
s1<s2 False
s1>=s2 True
s1<=s2 False
```

program to overload multiple operator to work on Employee objects:

```
class employee:
    def __init__(s,name,salary):
        s.name=name
        s.salary=salary
    def __mul__(s,other):
        return s.salary*other.days
class timesheet:
    def __init__(s,name,days):
        s.name=name
        s.days=days
e=employee("sangam kumar",3000000)
t=timesheet("sangam kumar",9)
print("this month salary:", e*t)
```

output:

```
this month salary: 27000000
```

2). method overloading:

- if 2 method having same name but different type of arguments then those method are said to be overloaded methods.

example:

- m1(int a)
- m1(double d)
- but in python method overloading is not possible.
- if we are tryin to declare multiple methods with same name and different number of arguments then python will always consider only last method.

example:

```
class test:
   def m1(self):
      print("no arg method")
   def m1(s,a):
      print('one arg method')
   def m1(self,a,b):
      print("two-arg method")
t=test()
# t.m1()
# t.m2()
t.m1(25,25)
```

output:

```
two-arg method
```

- in the above program python will only last method
- how we can handle overloaded method in python
- most of the times, if method with variable number of arguments required then we can handle with default arguments or with variable or with variable number of arguments method.

program with default arguments:

```
class test:
   def sum(s,a=None,b=None,c=None):
      if a!=None and b!=None and c!=None:
         print("the sum of 3 numbers:",a+b+c)
      elif a!=None and b!=None:
         print("the sum of 2 numbers:",a+b)
      else:
         print("please provide 2 or 3 arguments")
t=test()
t.sum(10,20)
t.sum(10,20,30)
t.sum(30)
```

output:

```
the sum of 2 numbers: 30
the sum of 3 numbers: 60
please provide 2 or 3 arguments
```

Example:

```
class test:
   def sum(s,*a):
      total=0
      for x in a:
         total=total+x
         print("the sum=",total)
t=test()
t.sum(10,20)
t.sum(10,20,30)
t.sum(10)
t.sum()
```

output:

```
the sum= 10
the sum= 30
the sum= 10
the sum= 30
the sum= 60
the sum= 10
```

3) constructor overloading:

- constructor overloading is not possible in python.
- if we define multiple constructors then last constructor will be considered.

Example:

```
class test:
  def __init__(s):
    print("no-arg constructor")
  def __init__(s,a):
    print("one-arg constructor")
  def __init__(self,a,b):
    print("Two arg constructor")
# t1=test()
# t1=test(20)
t1=test(10,20)
```

output:

- Two arg constructor
- in the above program only two arg constructor is available.
- but based on our requirement we can declare constructor with default arguments and variable number of arguments.

constructor with default arguments:

example:

```
class test:
  def __init__(s,a=None,b=None,c=None):
    print("constructor with 0 | 1 | 2 | 3 numbers of arguments")
t1=test()
t2=test(10)
t3=test(10,20)
t4=test(10,20,30)
```

output:

```
constructor with 0 | 1 | 2 | 3 numbers of arguments
constructor with 0 | 1 | 2 | 3 numbers of arguments
constructor with 0 | 1 | 2 | 3 numbers of arguments
constructor with 0 | 1 | 2 | 3 numbers of arguments
```

- constructor with variable number of arguments:

example:

```
class test:
  def __init__(s,*a):
    print("constructor with variable number of arguments")
t1=test()
```

```
t2=test(10)
t3=test(10,20)
t4=test(10,20,30)
t5=test(10,20,30,40,50,60,70)
```

output:

```
constructor with variable number of arguments
constructor with variable number of arguments
constructor with variable number of arguments
constructor with variable number of arguments
constructor with variable number of arguments
```

3) overriding:

method overriding:

- whatever members available in the parent class are by default available to the child class through inheritance if the child class not satisfied with parent class
- implementation then chid class is allowed to redefine that method in the child class based on its requirement. this concept is called overriding.
- overriding concept applicable for both methods and constructors.

program for method overriding:

```
class p:
   def property(s):
      print("gold+land+cash+power")
   def marry(s):
      print("skills center")
class c(p):
   def marry(s):
      print("t3")
r=c()
r.property()
r.marry()
```

output:

```
gold+land+cash+power
t3
```

- from overriding method of child class we can call parent class method also by using super() ,method.

Example:

```
class p:
   def property(s):
      print("Gold+Land+Cash+Power")
   def marry(s):
      print("skills center")
class c(p):
   def marry(s):
      super().marry()
      print("t3")
r=c()
r.property()
r.marry()
```

output:

```
Gold+Land+Cash+Power
skills center
t3
```

program for constructor overriding:

```
class p:
   def __init__(s):
      print("parent constructor")
class c(p):
   def __init__(self):
      print("child constructor")
r=c()
```

output:

```
child constructor
```

- in the above example, if child class does not contain constructor, then parent class constructor will be executed
- from child class constructor we can call parent class constructor by using super() method.
- program to call parent class constructor by using super():

Example:

```
class person:
   def __init__(s,name,age):
      s.name=name
      s.age=age
class employee(person):
   def __init__(s,name,age,eno,esal):
      super().__init__(name,age)
      s.eno=eno
      s.esal=esal
   def display(s):
      print("Employee Name:",s.name)
      print("Employee Age:",s.age)
      print("Employee Number:",s.eno)
      print("Employee Salary:",s.esal)
e1=employee("Sangam Kumar",33,5624,3300)
e1.display()
e2=employee("Satayam Kumar",39,4424,6600)
e2.display()
e3=employee("Sujeet Kumar",53,4524,9900)
e3.display()
```

output:

```
Employee Name: Sangam Kumar
Employee Age: 33
Employee Number: 5624
Employee Salary: 3300
Employee Name: Satyam Kumar
Employee Age: 39
Employee Number: 4424
```

Employee Salary: 6600
Employee Name: Sujeet Kumar
Employee Age: 53
Employee Number: 4524
Employee Salary: 9900

ABSTRACT METHOD

- Abstraction in Python is a concept that allows you to hide complex implementation details and show only the necessary features or functionality of an object or system.
- 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

❖ abstract method:

- sometimes we don't know about implementation, still we can declare a method such types of methods are called abstract methods, i.e abstract method has only declaration but not implementation.
- in python we can declare abstract method by using @abstractmethod decorator as follows

```
@abstractmethod
def m1(self):
   pass
```

@abstractmethod decorator present in abc module hence compulsory we should import abc module, otherwise we will get error

abc -->abstract base class module

Example:

```
class test:
   @abstractmethod
   def m1(self):
      pass
```

output:

NameError: name 'abstract method' is not defined

example:

```
from abc import*
class test:
   @abstractmethod
   def m1(self):
      pass
from abc import*
class fruit:
   @abstractmethod
   def test(self):
      pass
```

- child classes are responsible to provide implementation for parent abstract method.

❖ Abstract class:

- some times implementation of a class is not complete such type of partially implementation classes are called abstract classes every abstract class in python should be derived from ABC class which is present in abc module.

case-1

```
from abc import*
class test:
   pass
t=test()
```

- in the above code we can create object for test class because it is concrete class and it does not contain any abstract method.

case-2:

```
from abc import*
class test(ABC):
   pass
t=test()
```

- in the above code we can create object even it is derived from ABC class because it does not contain any abstract method.

case-3

```
from abc import*
class test(ABC):
   @abstractmethod
   def m1(self):
      pass
t=test()
TypeError: can't instantiate abstract class test with abstract method m1
```

case-4:

```
from abc import*
class test:
   @abstractmethod
   def m1(self):
      pass
t=test()
```

- we can create object even class contains abstract method because we are not extending ABC class.

case-5

```
from abc import*
class test:
   @abstractmethod
   def m1(self):
      print("skills center")
t=test()
t.m1()
```

output:

```
skills center
```

- **conclusion:** if a class contains at least one abstract method and if we are extending ABC class then instantiation is not possible

"Abstract class with abstract method instantiation is not possible"

- parent class abstract methods should be implemented in the child classes. otherwise, we cannot instantiate child class. if we are not creating child class then we won't get any error.

Example:

```
from abc import*
class vehicle(ABC):
   @abstractmethod
   def noofwheels(self):
      pass
class bus(vehicle):
   pass
```

- it is valid because we are not creating child class object.

case-6

```
from abc import*
class vehicle(ABC):
   @abstractmethod
   def noofwheels(self):
      pass
class bus(vehicle):
   pass
b=bus()
```

output:

```
TypeError: Can't instantiate abstract class bus with abstract method noofwheels
```

- if we are extending abstract class and does not override its abstract method then child class is also abstract and instanton is not possible.

Example:

```
from abc import*
class vehicle(ABC):
   @abstractmethod
   def noofwheels(self):
      pass
class bus(vehicle):
   def noofwheels(self):
      return 7
class auto(vehicle):
   def noofwheels(self):
      return 3
b=bus()
print(b.noofwheels())#7
a=auto()
print(a.noofwheels())#3
```

output:

```
7
3
```

interfaces in python:

- in general, if an abstract class contains only abstract methods such type of abstract class is considered as interface

example:

```
from abc import*
class dbinterface(ABC):
   @abstractmethod
   def connect(self):
      pass
   @abstractmethod
   def disconnect(self):
      pass
class oracle(dbinterface):
   def connect(self):
      print('connecting to oracle database.')
   def disconnect(self):
      print("disconnecting to oracle database.")

class sybase(dbinterface):
   def connect(self):
      print("connecting to sybase database.")
   def disconnect(Self):
       print("disconnecting to sybase database.")
dbname=input("Enter Database Name:")
classname=globals()[dbname]
x=classname()
x.connect()
x.disconnect()
```

output:

```
Enter Database Name:oracle
connecting to oracle database.
disconnecting to oracle database.

Enter Database Name:sybase
connecting to sybase database.
disconnecting to sybase database.
```

Note: the inbuilt function globals()[str] converts the string 'str' into a class name and returns the classname.

program: Reading class name from the file.

program: Reading class name from the file.

raj.txt #EPSON

raj1.txt #HP

```
from abc import*
class printer(ABC):
   @abstractmethod
```

```
    def printit(self,text):
        pass
    @abstractmethod
    def disconnect(self):
        pass
class EPSON(printer):
    def printit(self,text):
        print("printing from EPSON printer.")
        print(text)
    def disconnect(self):
        print("printing completed on EPSON printer...")
class HP(printer):
    def printit(self,text):
        print("printing from HP printer.")
        print(text)
    def disconnect(self):
        print("printinf completed on HP printer..")
with open('raj.txt','r') as f:
#with open('raj1.txt','r') as f:
    pname=f.readline()
classname=globals()[pname]
x=classname()
x.printit("This data has to print...")
x.disconnect()
```

output:

```
printing from EPSON printer.
This data has to print...
printing completed on EPSON printer...

printing from HP printer.
This data has to print...
printing completed on HP printer..
```

CONCREATE CLASS VS ABSTRACT CLASS VS INTERFACE

1. if we don't know anything about implementation just, we have requirement specification then we should go for interface.
2. If we are talking about Implementation but not completely then we should go for abstract class. (Partially implemented class)
3. If we are talking about implementation completely and provide service then we should go for concrete class.

Example:

```
from abc import*
class collegeautomation(ABC):
    @abstractmethod
    def m1(self):
        pass
    @abstractmethod
    def m2(self):
        pass
    @abstractmethod
    def m3(self):
        pass
class Abscls(collegeautomation):
    def m1(self):
        print("m1 method implementation")
    def m2(self):
        print("m2 method implementation")
class concreatecls(Abscls):
    def m3(self):
        print("m3 method implementation")
r=concreatecls()
r.m1()
r.m2()
r.m3()
```

Output:

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

PUBLIC, PROTECTED AND PRIVATE ATTRIBUTES:

- By default, every attribute is public. We can access from anywhere from anywhere either within the class or from outside of the class.

Example:

name= 'skills'

- Protected attributes can be accessed within the class anywhere but from outside of the class only in child classes. We can specify an attribute as protected by prefixing with _ symbol.

Syntax:

_varaiblename=value

Example:

_name= 'skills'

- But is just convention and in reality, does not exist protected attributes.
- Private attribute can be accessed only within the class i.e from outside of the class we can not access. We can declare a variable as private explicitly by prefixing with 2 __ underscore symbols.

Syntax:

__varaiblename=value

Example:

__name= "skills"

Program:

```
class test:
   x=30
   _y=40
   __z=50
   def m1(self):
      print(test.x)
      print(test._y)
      print(test.__z)
r=test()
r.m1()
print(test.x)
print(test._y)
print(test.__z) #AttributeError: type object 'test' has no attribute '__z'
```

output:

```
30
40
50
30
40
AttributeError: type object 'test' has no attribute '__z'
```

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

We can not access private variables directly from outside of the class.

But we can access indirectly as follows:

Objectreference._classname__variablename

Example:

```
class test:
   def __init__(self):
      self.__x=30
r=test()
print(r._test__x)
```

Output:

30

❖ __str__()method:

- Whenever we are printing any object reference internally __str__() method will be called which is returns string in the following format.

➢ __main__.classname object at 0X02214B0

- To return meaningful string representation we have to override __str__() method.

Example:

```
class student:
   def __init__(self,name,rollno):
      self.name=name
      self.rollno=rollno
   def __str__(self):
      return 'this is student with Name:{} and Rollno:{}'.format(self.name,self.rollno)
s1=student("sangam kumar",103)
s2=student("satyam kumar",106)
print(s1)
print(s2)
```

output:

```
this is student with Name:sangam kumar and Rollno:103
this is student with Name:satyam kumar and Rollno:106
```

Difference between str() repr()

or

Difference between __str__() and __repr__()

- Str() internally calls __str__ function and hence functionality of both is same.
- Similarly, repr() internally calss __repr__() function and hence functionality of both is same.
- Str() returns a string containing a nicely printable representation object.
- The main purpose of str() is for readability. It may not possible to convert result string to original object.

Example:

```
import datetime
today=datetime.datetime.now()
s=str(today)#converting datetime object to str
print(s)
d=eval(s)#converting str object to datetime
```

Output:

```
2023-09-11 00:29:09.529639
Traceback (most recent call last):
SyntaxError: leading zeros in decimal integer literals are not permitted;
use an 0o prefix for octal integers
```

But repr() returns a string containing a printable representation of object.
The main goal of repr() is unambiguous. We can convert result string to original object by using eval() function, which may not possible in str() function.

Example:

```
import datetime
today=datetime.datetime.now()
s=repr(today)#converting datetime object to str
print(s)
d=eval(s)#converting str object to datetime
print(d)
```

output:

```
datetime.datetime(2023, 9, 11, 0, 36, 36, 584141)
2023-09-11 00:36:36.584141
```

Note:
It is recommended to use repr() instead of str()

Mini Project:

- 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
```

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 Exception?

- An unwanted and unexpected event that disturbs normal flow of program is called 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

❖ 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
```

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()
```

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.
- Follow these step-by-step instructions:

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.

T3 SKILLS CENTER

❖ 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 (इनोवेशन) 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।

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

TWKSAA RID MISSION

(Research)	(Innovation)	(Discovery)
अनुसंधान करने के महत्वपूर्ण कारण:	**नवीनीकरण करने के महत्वपूर्ण कारण:**	**खोज करने के महत्वपूर्ण कारण:**
1. नई ज्ञान की प्राप्ति	1. प्रगति के लिए	1. नए ज्ञान की प्राप्ति
2. समस्याओं का समाधान	2. परिवर्तन के लिए	2. ज्ञान के विकास में योगदान
3. तकनीकी और व्यापार में उन्नति	3. उत्पादन में सुधार	3. अविष्कारों की खोज
4. विकास को बढ़ावा देना	4. प्रतिस्पर्धा में अग्रणी होने के लिए	4. समस्याओं का समाधान
5. सामाजिक प्रगति	5. समाज को लाभ	5. समाज के उन्नति का माध्यम
6. देश विज्ञान और प्रौद्योगिकी का विकास	6. देश विज्ञान और प्रौद्योगिकी के विकास।	6. देश विज्ञान और तकनीक के विकास

➢ जो लोग रिसर्च, इनोवेशन और डिस्कवरी करते हैं उन लोगों को ही हमें अपना नायक, प्रतीक एवं आदर्श मानना चाहिए क्योंकि यें लोग हमारे समाज, देश एवं विज्ञान के क्षेत्र में प्रगति, विकास और समस्याओं के समाधान में महत्वपूर्ण भूमिका निभाते हैं।

मैं राजेश प्रसाद एक वीणा उठाया हूँ Research, Innovation and Discovery का 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 ही हो |

"अगर आप भी Research, Innovation and Discovery के क्षेत्र में रूचि रखतें हैं एवं अपनी प्रतिभा से दुनियां को कुछ नया देना चाहतें तो हमारे इस त्वक्सा रीड मिशन (TWKSAA RID MISSION) से जरुर जुड़ें" |

- **राजेश प्रसाद**