PYTHON

LECTURE 38

Today's Agenda

- **Introduction To Object Oriented Programming-I**
  - Problems With Procedure Oriented Programming
  - What Is Object Oriented Programming ?
  - What Is A Class ?
  - What Is An Object ?
  - Syntax Of Creating A Class In Python
  - Syntax Of Creating Object
  - Types Of Data Members A Class Can Have
  - The Method __**init__()**
  - The Argument **self**
  - Passing Parameters To __**init__()**

Question ???

- **Can you tell , what kind of programming paradigm we have followed this point in Python ?**

- The answer is : **POP** (**Procedure Oriented Programming**)

  - In all the programs we wrote till now, we have designed our program around **functions** i.e. **blocks of statements which manipulate data**.

  - This is called the ***procedure-oriented programming***.

Advantages

- **Advantages Of Procedure Oriented Programming**
  - It's **easy** to implement
  - The ability to **re-use the same code** at different places in the program **without copying it**.
  - An easier way to **keep track** of program flow **for small codes**
  - Needs **less memory**.

Disadvantages

- **Disadvantages Of Procedure Oriented Programming**

  - **Very difficult** to relate with **real world objects**.

  - **Data** is **exposed** to whole program, so **no security for data**.

  - **Difficult** to create **new data types**

  - Importance is given to the **operation on data** rather than **the data**.

So , What Is The Solution ?

- Solution to all the previous **4 problems** is **Object Oriented Programming**

- Many people consider **OOP** to be a modern programming paradigm, but the roots go back to **1960**s.

- The **first programming language to use objects** was **Simula 67**

What Is OOP?

- **OOP** is a **programming paradigm** (***way of developing programs***)

- In **OOP**, we have the **flexibility** to represent **real-world objects** like **car**, **animal**, **person**, **ATM** etc. in our code

- It allows us to **combine** the **data** and **functionality** and **wrap it inside** something which is called an **object**

What Is An Object?

- In programming **any real world entity** which has specific **attributes** or **features** can be represented as an **object**.

- In simple words, an **object** is something that possess some **characteristics** and can **perform certain functions**.

What Is An Object?

- For example, **car** is an **object** and can perform **functions** like **start**, **stop**, **drive** and **brake**.
- These are the **functions** or **behaviours** of a car.
- And the **characteristics** or **attributes** are **color** of car, **mileage**, **maximum speed**, **model** , **year** etc.

Are We Objects ?

- Yes , **we humans** are **objects** because:
  - We have **attributes** as **name**, **height**, **age** etc.
  - We also can show **behaviors** like **walking**, **talking**, **running**, **eating** etc

Classes

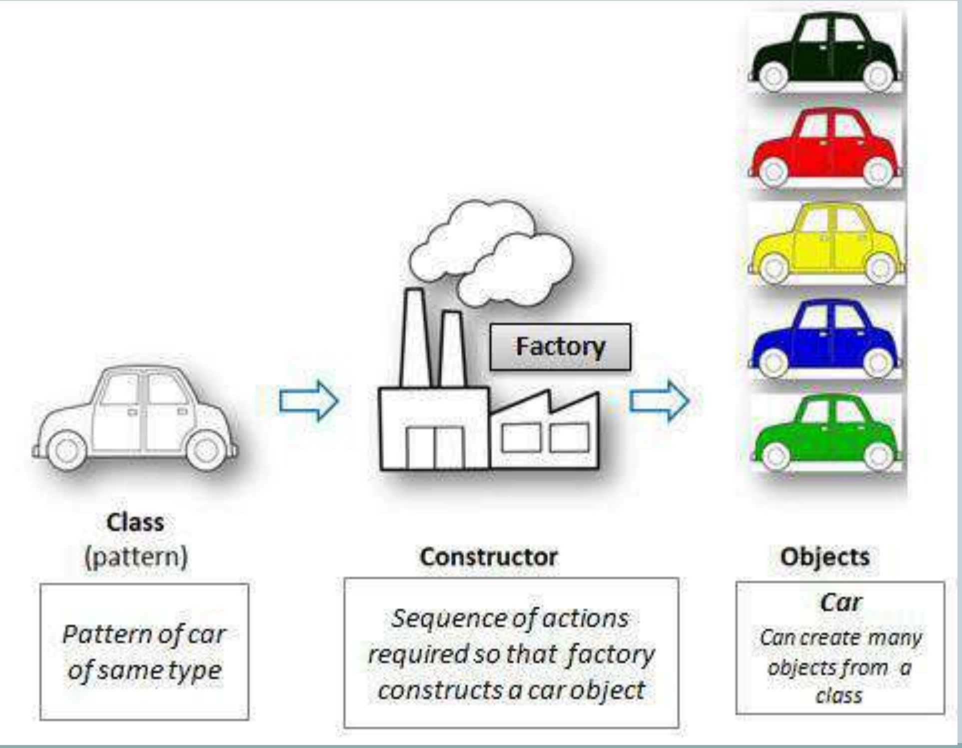
- Now to **create/represent** objects we first have to write all their **attributes** and **behaviours** under a **single group** .

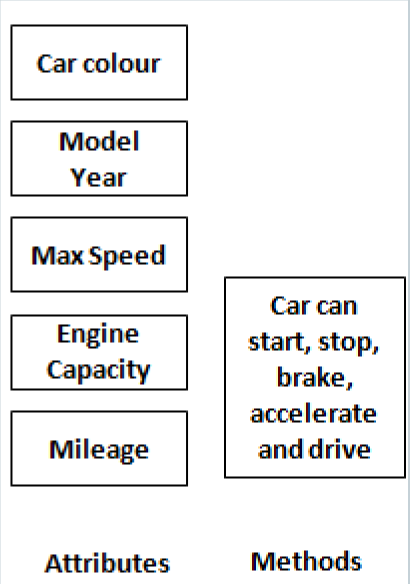
- This **group** is called a **class** .

- Thus a class is an **architecture/blueprint** of the object. It is a **proper description** of the **attributes** and **methods** of the object.

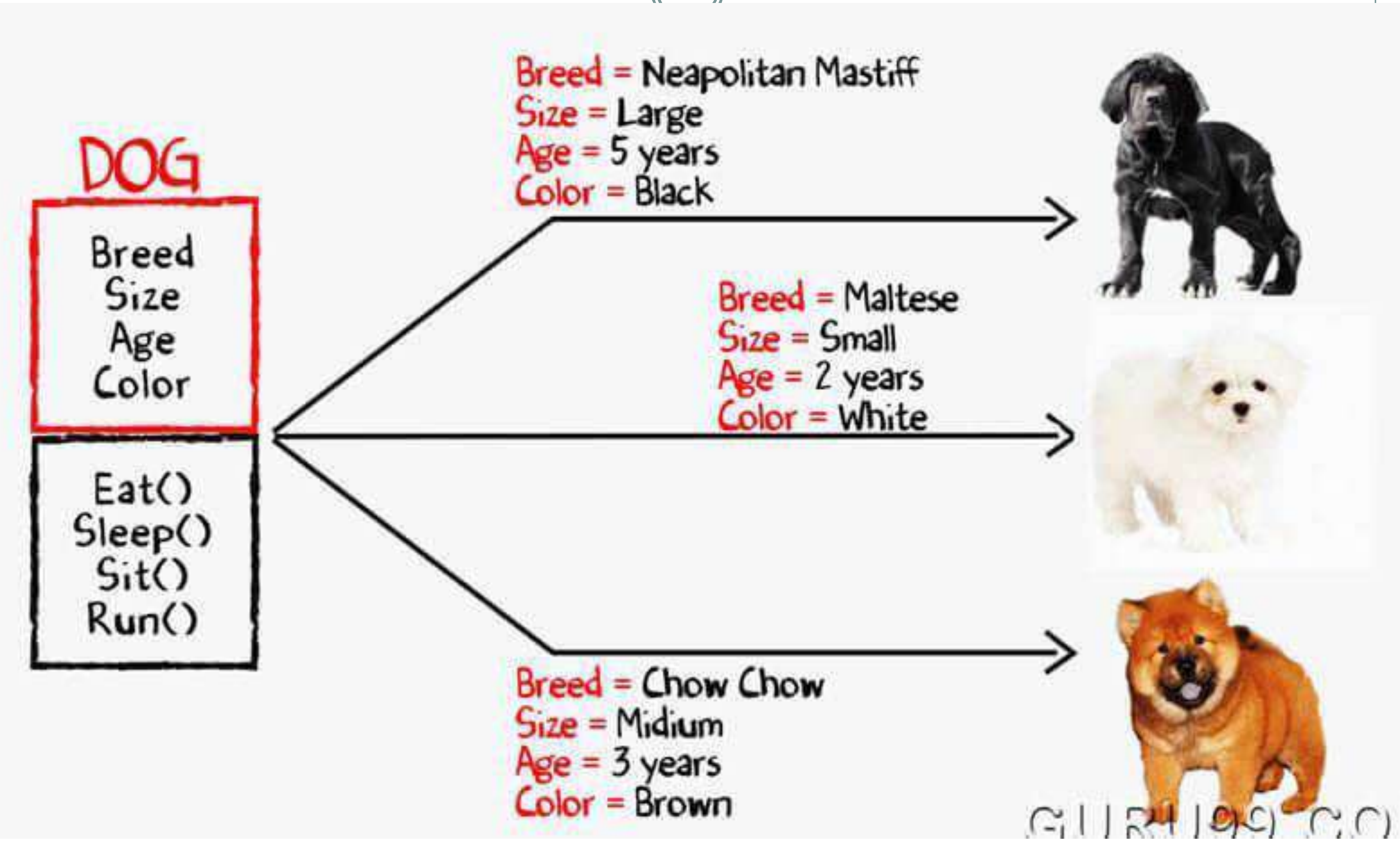
Classes

- **For Example:-** The design of a **car** of same type is a **class**. We can create **many objects** from a **class**. Just like we can make **many cars** of the same type from a **design** of **car**.



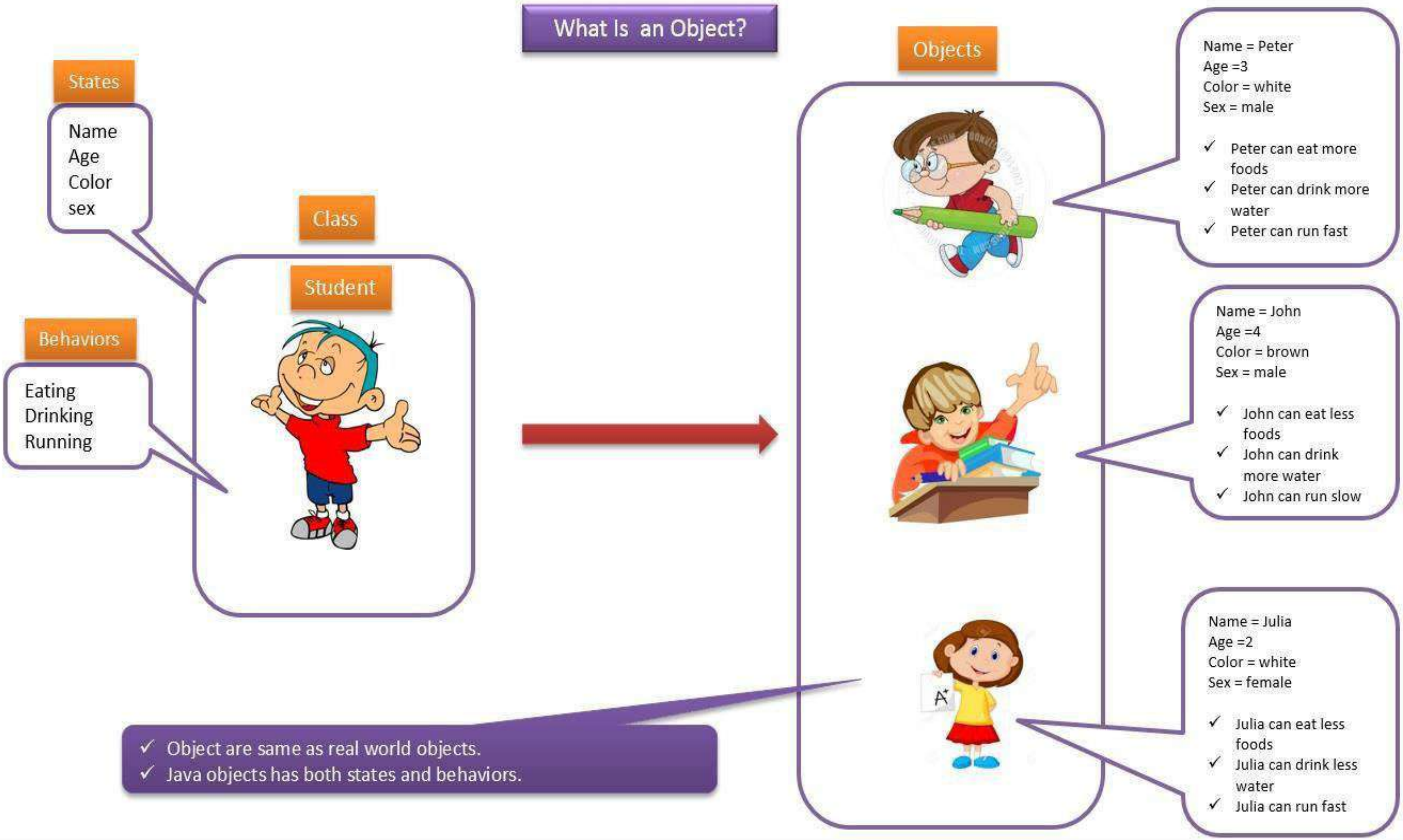


A Dog Class



DOG
Breed
Size
Age
Color
Eat()
Sleep()
Sit()
Run()
Breed = Neapolitan Mastiff
Size = Large
Age = 5 years
Color = Black
Breed = Maltese
Size = Small
Age = 2 years
Color = White
Breed = Chow Chow
Size = Midium
Age = 3 years
Color = Brown

A Student Class



Creating A Class

- Defining a class is simple in **Python**.

- We start with the **class keyword** to indicate that we are creating a class, then we add the name of the class followed by a **colon**

- We can then add **class members** , which are **methods** and **attributes**

Syntax Of Creating A Class

Syntax:

```
class <class_name>:
    # class members
```

Example:

```
class Emp:
    pass
```

Creating Objects

- In order to **use** a **class** we have to create it's object which is also called **instantiating** a class because **objects** are also called **instance** of the class

- So, to create an **instance** of a class, we use the **class name**, followed by **parentheses** and assign it to a **variable**.

Syntax Of Creating Object

Syntax:

```
var_name=class_name( )
```

Example:

```
e=Emp()
```

Full Code

```python
class Emp:
   pass

e=Emp()
print(type(e))
print(e)
```

Output:

```
<class '__main__.Emp'>
<__main__.Emp object at 0x000000002CC8860>
```

1. The first line shows the **class name** which is **Emp**.

2. The second line shows the **address** of the **object** to which the reference **e** is **pointing**

3. The name ____**main**___ is the name of the **module** which Python automatically allots to our file

Adding
Data Members/Attributes

- Once we have defined the class , our next step is to provide it data members/variables which can be used to hold values related to objects.

- In **Python** , a class can have **3 types** of variables:

  - **Instance Variables: Created per instance basis**

  - **Local Variables: Created locally inside a method and destroyed when the method execution is over**

  - **Class Variables: Created inside a class and shared by every object of that class. Sometimes also called as static variables**

What Is An Instance Variable?

- **Object variables** or **Instance Variables** are created by **Python** for **each individual object** of the class.

- In this case, ***each object has its own copy of the instance variable*** and they are not shared or related in any way to the field by the same name in a different object

Creating Instance Variables

- Creation of **instance variables** in **Python** is **entirely different** than **C++** or **Java**

- In these languages , we declare the **data members** inside the class and when we **instantiate** the class , these members are **allocated space** .

Creating Instance Variables In C++

- For example in **C++** ,we would write :

```
class Emp
{
int age;
char name[20];
double salary;
........
........
};
```

These are called **instance variables** in C++

Now to use this **Emp** class we would say:

Emp obj;

Doing this will create an **object** in memory by the name **e** and will contain three **instance members** called as **age** , **name** and **salary .** Also this line will **automatically call** a special method called <u>**constructor**</u> for **initializing the object**

Creating Instance Variables In Java

- In **Java** ,we would write :

```
class Emp
{
int age;
String name;
double salary;
........
........
}
```

These are called **instance variables** in Java

Now to use this **Emp** class we would say:

Emp obj=new Emp( );

Doing this will create an **object** in **heap** with the **data members** as **age** , **name** and **salary** and the **reference e** will be pointing to that **object.** Here also the special method called <u>**constructor**</u> will be called **automatically** for **initializing the object**

Creating Instance Variables In Python

- But in Python we use a very special method called **__init__()** , to **create** as well as **initialize** an object's initial attributes by giving them their **default value**.

- Python calls this method **automatically** , as soon as the object of the class gets created.

- Since it is called **automatically** , we can say it is like a **constructor** in **C++** or **Java**.

Full Code

```
class Emp:
   def __init__(self):
        print("Object created. . .")

e=Emp()
```

Output:

```
Object created. . .
```

As you can observe , **Python** has **automatically called** the special method **__init__()** as soon as we have created the object of **Emp** class

Another Example

```
class Emp:
   def __init__(self):
         print("Object created. . .")

e=Emp()
f=Emp()
g=Emp()
```

Output:

```
Object created. . .
Object created. . .
Object created. . .
```

The argument **self** ?

- You must have noticed that the code is using an argument called **self** in the argument list of **__init__()**

- So , now **2 questions** arise , which are :
  - **What is self ?**
  - **Why it is required ?**

What Is **self** ?

- In **Python** , whenever we create an object , **Python** calls the method __**init()__**
- But while calling this method , **Python** also passes the **address of the object** , for which it is calling __**init__()** , as the **first argument**.
- Thus , when we define the __**init__()** method we must provide it **atleast one formal argument** which will receive the object's address .
- This argument is named as **self**

What If We Don't Create self ?

```
class Emp:
   def __init__():
        print("Object created. . .")

e=Emp()
```

As you can observe , Python has generated an exception , since it has passed the object address as argument while calling the method __init__() but we have not declared any argument to receive it

Output:

```
    e=Emp()
TypeError: __init__() takes 0 positional arguments but 1 was given
```

Can We Give Some Other Name To **self** ?

```
class Emp:
    def __init__(myself):
        print("Object created. . .")

e=Emp()
```

As you can observe , Python has allowed us to use the name myself instead of self , but the convention is to always use the word self

Output:

```
Object created. . .
```

More About **self**

- Python always passes **address of the object** to every **instance method** of our class whenever we call it, not only to the method **__init__()**

- So, every **instance method** which we define in our class has to compulsorily have atleast one argument of type **self**

More About **self**

- The argument **self** always **points** to the **address of the current object**

- We can think it to be like **this reference** or **this pointer** of **Java** or **C++** languages

Is **self** A Keyword ?

- **No , not at all**
- Many programmers wrongly think **self** to be a **keyword** but it is not so.
- It is just a name and can be changed to anything else but the convention is to always use the name **self**
- **Another Important Point!**
- The argument **self** is **local** to the method body , so we cannot use it outside the method

Guess The Output

```
class Emp:
   def __init__(self):
        print("Object Created...")



e=Emp()
print(self)
```

Output:

```
    print(self)
NameError: name 'self' is not defined
```

The Most Important Role Of **self**

- We can also use **self** to **dynamically** add **instance members** to the **current object**.

- To do this ,we simply have to use **self** followed by **dot operator** followed by **name** of the variable along with it's **initial value**

- **Syntax:**

```
class <class_name>:
   def __init__(self):
         self.<var_name>=value
         .
         .
```

Example

```
class Emp:
    def __init__(self):
        self.age=25
        self.name="Rahul"
        self.salary=30000.0

e=Emp()
print("Age:",e.age,"Name:",e.name,"Salary:",e.salary)
```

The variables self.age,self.name and self.salary are called instance variables

Remember , we cannot use self outside the class . So outside the class we will have to use the reference variable e

Another very important point to understand if you are from C++ background is that in Python by default everything in a class is public . So we can direclty access it outside the class

Output:

```
Age: 25 Name: Rahul Salary: 30000.
```

A Very Important Point!

- The **instance variables** called **age , name** and **salary** are accessed in **2 ways** in **Python**:
  - Inside the methods of the class , they are always accessed using **self** so that **Python** will refer them for **current object**
  - Outside the class , we cannot access them using **self** because ***self is only available within the class***.
  - So outside the class we have to access them using the **object reference** we have created

Guess The Output ?

```
class Emp:
  def __init__(self):
      self.age=25
      self.name="Rahul"

e=Emp()
e.salary=30000.0
print("Age:",e.age,"Name:",e.name,"Salary:",e.salary)
```

Unlike C++ or Java , in Python we can create instance variables outside the class by directly using the object reference

Output:

```
Age: 25 Name: Rahul Salary: 30000.0
```

A Problem With The Code

- Although the code works fine , but it has one problem .

- The problem is that for **every object** of **Emp** class , **Python** will call **__init__()** method and thus every object will be **initialized** with the **same values**

- To overcome this problem we can make the method **__init__()** parameterized

Passing Parameters To __init__()

- Since __init__() is also a method so just like other methods we can pass **arguments** to it .

- But we need to remember 2 things for this:

  - **Since __init__() is called by Python at the time of object creation so we will have to pass these arguments at the time of creation of the object**

  - **We will have to define parameters also while defining __init__() to receive these arguments**

- Finally using these **parameters** we can **initialize** instance members to **different values** for **different objects**

Passing Parameters To __init__()

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

e=Emp(25,"Rahul",30000.0)
print("Age:",e.age,"Name:",e.name,"Salary:",e.salary)
f=Emp(31,"Varun",45000.0)
print("Age:",f.age,"Name:",f.name,"Salary:",f.salary)
```

The variables age, name an salary are called local variables

Output:

```
Age: 25 Name: Rahul Salary: 30000.0
Age: 31 Name: Varun Salary: 45000.0
```

An Important Point

- The argument **self** , should always be the first argument as **Python** passes the address of the current object as the first argument

- The **variables age** , **name** and **salary** used in the argument list of ___**init__()** are called **parameters** or **local variables**.

- They will only **survive** until the method is **under execution** and after that they will be **destroyed by Python**

An Important Point

- Any **variable** declared inside the body of any method inside the class without using **self** will also be called as **local variable**

- It is a **common convention** to give **parameters** the **same name** as **instance members** , but it is not at all compulsory.

Passing Parameters To __init__()

```
class Emp:
  def __init__(self,x,y,z):
     self.age=x
     self.name=y
     self.salary=z

e=Emp(25,"Rahul",30000.0)
print("Age:",e.age,"Name:",e.name,"Salary:",e.salary)
f=Emp(31,"Varun",45000.0)
print("Age:",f.age,"Name:",f.name,"Salary:",f.salary)
```

Output:

```
Age: 25 Name: Rahul Salary: 30000.0
Age: 31 Name: Varun Salary: 45000.0
```

Guess The Output ?

```python
class Emp:
    def __init__(self,name):
            self.name=name
    def __init__(self,name,age):
            self.name=name
            self.age=age
    def __init__(self,name,age,sal):
            self.name=name
            self.age=age
            self.sal=sal
```

```python
e1=Emp("amit")
e2=Emp("sumit",23)
e3=Emp("deepak",34,50000.)
print(e1.name)
print(e2.name,e2.age)
print(e3.name,e3.age,e3.sal)
```

Output:

```
    e1=Emp("amit")
TypeError: __init__() missing 2 required positional arguments: 'age' and 'sal'
```

Why Didn't The Code Run ?

- Recall , that we have already discussed that **PYTHON DOES NOT SUPPORT METHOD/FUNCTION OVERLOADING .**

- So if **two methods** have **same name** then the **second copy** of the method will **overwrite** the **first copy**.

- So , in our case **Python** remembers only one **__init__()** method , which is defined last and since it is taking **3 arguments** (excluding self) so our call:

  e1=Emp("amit")

  generated the exception

Question ?

- **Can we do something so that the code runs with different number of arguments passed to Emp objects ?**

- **Yes !**

- The solution is to use **default arguments**

Solution

```
class Emp:
    def __init__(self,name,age=0,sal=0.0):
            self.name=name
            self.age=age
            self.sal=sal

e1=Emp("amit")
e2=Emp("sumit",23)
e3=Emp("deepak",34,50000.)
print(e1.name)
print(e2.name,e2.age)
print(e3.name,e3.age,e3.sal)
```

Output:

```
amit
sumit 23
deepak 34 50000.0
```

PYTHON

LECTURE 39

Today's Agenda

- **Introduction To Object Oriented Programming-II**
  - Types Of Methods
  - Adding Instance Methods
  - Obtaining Details Of Instance Variables
  - Different Ways To Create Instance Variables
  - Deleting Instance Variables

Adding
Methods In Class

- Once we have defined the class , our next step is to provide methods in it

- In **Python** , a class can have **3 types** of methods:

  - **Instance Methods: Called using object**

  - **Class Methods: Called using class name**

  - **Static Methods: Called using class name**

Adding Instance Methods

- **Instance methods** are the most common type of methods in Python classes.

- These are called **instance methods** because they can access **instance members** of the object.

Adding Instance Methods

- These methods always take **atleast one parameter**, which is normally called **self**, which points to the **current object** for which the method is called.
- Through the **self** parameter, **instance methods** can access **data members** and other methods on the same object.
- This gives them a lot of power when it comes to **modifying** an **object's state**.

Example

```
class Emp:
    def __init__(self,age,name,salary):
            self.age=age
            self.name=name
            self.salary=salary
    def show(self):
            print("Age:",self.age,"Name:",self.name,"Salary:",self.salary)

e=Emp(25,"Rahul",30000.0)
f=Emp(31,"Varun",45000.0)
e.show()
f.show()
```

Output:

```
Age: 25 Name: Rahul Salary: 30000.0
Age: 31 Name: Varun Salary: 45000.0
```

Exercise

- Write a program to create a class called **Circle** , having an instance member called **radius**. Provide following instance methods in your class:
  - **__init__() :** This method should initialize radius with the parameter passed
  - **cal_area():** This method should calculate and print the area of the Circle
  - **cal_circumference**(): This method should calculate and print the circumference of the Circle
- Finally , in the main script , create a **Circle** object **, initialize radius** with **user input** and calculate and display it's **area** and **circumference**

Output:

```
Enter radius:10
Area of circle is 314.1592653589793
Circumference of circle is 62.83185307179586
```

Solution

```
import math
class Circle:
    def __init__(self,radius):
            self.radius=radius
    def cal_area(self):
            area=math.pi*math.pow(self.radius,2)
            print("Area of circle is",area)
    def cal_circumference(self):
            circumf=math.tau * self.radius
            print("Circumference  of circle is",circumf)

radius=int(input("Enter radius:"))
cobj=Circle(radius)
cobj.cal_area()
cobj.cal_circumference()
```

Guess The Output ?

```
class Emp:

    def __init__(self):
        self.name="Amit"
        self.age=24
        self.sal=50000.0
    def show(self):
        print(age,name,sal)

e1=Emp()
e1.show()
```

Why did the code give exception?

The syntax we are using for accessing name , age and sal is only applicable to local variables and not for instance members.
And since there are no local variables by the name of name , age and sal , so the code is giving exception

Output:

```
    print(age,name,sal)
NameError: name 'age' is not defined
```

Guess The Output ?

```
class Emp:

    def __init__(self,name,age,sal):
            self.name=name
            self.age=age
            self.sal=sal
    def show(self):
            print(age,name,sal)

e1=Emp("amit",34,50000.0)
e1.show()
```

Why did the code give exception?

The variables name, age and sal are local variables declared inside the method `__init__()` and hence are not available to the method show() , so the code gave NameError exception

<u>Output:</u>

```
    print(age,name,sal)
NameError: name 'age' is not defined
```

Guess The Output ?

```
class Emp:

    def __init__(self):
        self.name="Amit"
        self.age=24
        self.sal=50000.0
    def show(self):
        print(self.age,self.name,self.sal)

e1=Emp()
e1.show()
```

Output:

```
24 Amit 50000.0
```

Obtaining Details Of Instance Variables

- **Every object** in **Python** has an **attribute** denoted by **__dict__.**

- This **attribute** is **automatically added by Python** and it contains all the **attributes** defined ***for the object itself***.

- It maps the **attribute name** to its **value**.

Guess The Output ?

```
class Emp:

    def __init__(self):
        self.name="Amit"
        self.age=24
        self.sal=50000.0


e1=Emp()
print(e1.__dict__)
```

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 50000.0}
```

Guess The Output ?

```
class Emp:

    def __init__(self):
            self.name="Amit"
            self.age=24
            sal=50000.0



e1=Emp()
print(e1.__dict__)
```

Output:

```
{'name': 'Amit', 'age': 24}
```

Guess The Output ?

```
class Emp:

    def __init__(self):
        self.name="Amit"
        self.age=24
    def set_sal(self):
        self.sal=50000.0

e1=Emp() print(e1.
__________dict__)
e1.set_sal()
print(e1.__dict__)
```

Output:

```
{'name': 'Amit', 'age': 24}
{'name': 'Amit', 'age': 24, 'sal': 50000.0}
```

Guess The Output ?

```
class Emp:
    def __init__(self):
            self.name="Amit"
            self.age=24
            self.sal=50000.0
    def show(self):
            print(self.name,self.age,self.sal,self.department)
e1=Emp()
print(e1.__dict__)
e1.__dict__['department']='IT'
print(e1.__dict__)
e1.show()
```

Since __dict__ is a dictionary , we can manipulate it and add/del instance members from it

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 50000.0}
{'name': 'Amit', 'age': 24, 'sal': 50000.0, 'department': 'IT'}
Amit 24 50000.0 IT
```

How Many Ways Are There To Create **Instance Variables** ?

- Till now we can say there are **4 ways** in **Python** to create **instance variables**:
  - Inside the **constructor**/**__init__()** method using **self**
  - Inside **any instance method** of the class using **self**
  - **Outside the class** using it's **object reference**
  - Using the instance attribute __**dict**__

Guess The Output ?

```
class Emp:
    def __init__(self,name,age,sal):
            self.name=name
            self.age=age
            self.sal=sal
    def setDept(self,department):
            self.department=department
    def setProject(self,project):
            self.project=project
    def setBonus(self,bonus):
            self.bonus=bonus
```

```
e1=Emp("Amit",24,30000.0)
e2=Emp("Sumit",34,45000.0)
e1.setDept("Finance")
e1.setProject("Banking Info System")
e1.setBonus(20000.0)
e2.setDept("Production")
print(e1.__dict__)
print()
print(e2.__dict__)
```

Since Python is dynamically typed language so object's of same class can have different number of instance variables

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 30000.0, 'department': 'Finance', 'project':
'Banking Info System', 'bonus': 20000.0}

{'name': 'Sumit', 'age': 34, 'sal': 45000.0, 'department': 'Production'}
```

Deleting Instance Variables

- We can **delete/remove** instance variables in 2 ways:
  - Using **del self .<var_name>** from the body of any **instance method** within the class
  - Using **del <obj_ref>.<var_name>** from **outside the class**

Guess The Output ?

```
class Boy:
    def __init__(self,name,girlfriend):
            self.name=name
            self.girlfriend=girlfriend
    def breakup(self):
            del self.girlfriend
b1=Boy("Deepak","Jyoti")
print(b1.__dict__)
b1.breakup()
print(b1.girlfriend)
```

Output:

```
{'name': 'Deepak', 'girlfriend': 'Jyoti'}
Traceback (most recent call last):
  File "classdemo7.py", line 10, in <module>
    print(b1.girlfriend)
AttributeError: 'Boy' object has no attribute 'girlfriend'
```

Guess The Output ?

```
class Engineer:
    def __init__(self,girlfriend,job):
            self.girlfriend=girlfriend
            self.job=job
    def fired(self):
            del self.job
e1=Engineer("Rani","Software Engineer")
print(e1.__dict__)
e1.fired()
del e1.girlfriend
print(e1.__dict__)
```

Output:

```
{'girlfriend': 'Rani', 'job': 'Software Engineer'}
{}
```

Guess The Output ?

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

e1=Emp("Amit",24,50000.0)
print(e1.__dict__)
del e1
print(e1.__dict__)
```

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 50000.0}
Traceback (most recent call last):
  File "classdemo8.py", line 11, in <module>
    print(e1.__dict__)
NameError: name 'e1' is not defined
```

Guess The Output ?

```
class Emp:
    def __init__(self,name,age,sal):
            self.name=name
            self.age=age
            self.sal=sal
    def remove(self):
            del self
e1=Emp("Amit",24,50000.0)
print(e1.__dict__)
e1.remove()
print(e1.__dict__)
```

Since the object pointed by self is also pointed by e1 , so Python didn't remove the object , rather it only removes the reference self

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 50000.0}
{'name': 'Amit', 'age': 24, 'sal': 50000.0}
```

Guess The Output ?

```
class Emp:
    def __init__(self,name,age,sal):
            self.name=name
            self.age=age
            self.sal=sal
e1=Emp("Amit",24,50000.0)
e2=Emp("Sumit",25,45000.0)
print(e1.__dict__)
print(e2.__dict__)
del e1.sal
del e2.age
print()
print(e1.__dict__)
print(e2.__dict__)
```

Since instance variables have a separate copy created for every object , so deleting an instance variable from one object will not effect the other object’s same instance variable

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 50000.0}
{'name': 'Sumit', 'age': 25, 'sal': 45000.0}

{'name': 'Amit', 'age': 24}
{'name': 'Sumit', 'sal': 45000.0}
```

PYTHON

LECTURE 40

Today’s Agenda

- **Introduction To Object Oriented Programming-III**

  - Adding Class Variables
  - Different Ways To Create A Class Variable
  - Different Ways To Access A Class Variable
  - Obtaining Details Of Class Variables
  - Deleting Class Variables

Class Variables

- **Class variables** are those variables which are defined within the **class body** **outside any method**

- They are also called as **static variables** , although there is no **static** keyword used with them

Class Variables

- The are **shared by all instances** of the class and **have the same value** for each instance of the class.

- They have a **single copy** maintained at the **class level**

What Is **Class Level** ?

- The term **class level** means inside the **class object**.

- In **Python** , ***for every class one special object is created called as*** **<u>class object</u>**

- **Don't think it is the same object which we create. No it is not that!**

- Rather , for every class , **Python** itself creates an object called as **class object** and inside this object all the **class / static** variables live

When Should We Use **Class Variable** ?

- Whenever we don't want to create a **separate copy** of the **variable** for **each object** , then we can declare it as a **class variable.**

- For example :

  - The variable **pi** in a class called **Circle** can be declared as a **class level variable** since all **Circle objects** will have the **same value** for **pi**

  - Another example could be a variable called **max_marks** in a class called **Student** . It should also be declared at the **class level** because each **Student** will have same **max_marks**

Using Class Variable

- We can use a class variable at **6 places** in **Python**:
  - Inside the **class body** but **outside any method**
  - Inside the **constructor** using the **name of the class**
  - Inside **instance method** using **name of the class**
  - Inside **classmethod** using **name of the class** or using the special reference **cls**
  - Inside **staticmethod** using the **name of the class**
  - From outside the class using **name of the class**

Declaring Inside Class Body

- To declare a **class variable** inside class body but outside any method body , we simply declare it below the **class header**

- **Syntax:**

```
class <class_name>:
    <var_name>=<value>
def __init__(self):
    // object specific code
```

This is called a **class variable**

- To access the **class level variables** we use **class name** before them with **dot operator**

How To Access and Modify Class Variables?

- We must clearly understand the difference between **accessing** and **modifying** .

- **Accessing** means we are just reading the value of the variable

- **Modifying** means we are changing it’s value

How To Access Class Variables?

- The **class variables** can be **accessed** in **4** ways:
  - Using **name of the class** anywhere in the program
  - Using **self** inside any **instance method**
  - Using **object reference** outside the class
  - Using special reference **cls** inside **classmethod**

How To Modify Class Variables?

- The **class variables** can be **modified** in **3** ways:
  - Using **name of the class** anywhere inside the methods of the class
  - Using special reference **cls** inside **classmethod**
  - Using **name of the class** outside the class body
- **Special Note:** We must never **modify** a **class variable** using **self** or **object reference** , because it will not **modify** the **class variable** , rather will create a new **instance variable** by the same name

Example

```
class CompStudent:
    stream = 'cse'
    def __init__(self,name,roll):
        self.name = name
        self.roll = roll

obj1 = CompStudent('Atul',1)
obj2 = CompStudent('Chetan', 2)
print(obj1.name)
print(obj1.roll)
print(obj1.stream)
print(obj2.name)
print(obj2.roll)
print(obj2.stream)
print(CompStudent.stream)
```

The variable **stream** is class variable

Everytime we will access the class variable **stream** from any object , the value will remain same

Output:

```
Atul
1
cse
Chetan
2
cse
cse
```

Exercise

- Write a program to create a class called **Emp** , having 3 **instance members** called **name** , **age** and **sal** . Also declare a **class variable** called **raise_amount** to store the **increment percentage** of **sal** and set it to **7.5** .
- Now provide following methods in your class
  - **__init__() :** This method should initialize instance members with the parameter passed
  - **increase_sal():** This method should calculate the increment in sal and add it to the instance member sal
  - **display**(): This method should display name , age and sal of the employee
- Finally , in the main script , **create 2 Emp objects** , **initialize them** and **increase their salary** . Finally **display** the data

Output:

```
Before incrementing :
_____________________
Amit 24 50000.0
Sumit 26 45000.0

After incrementing by 7.5 percent:
__________________________________
Amit 24 53750.0
Sumit 26 48375.0
```

Solution

```
class Emp:
    raise_amount=7.5
    def __init__(self,name,age,sal):
        self.name=name
        self.age=age
        self.sal=sal
    def increase_sal(self):
        self.sal=self.sal+(self.sal*Emp.raise_amount/100)
    def display(self):
        print(self.name,self.age,self.sal)

e1=Emp("Amit",24,50000.0)
e2=Emp("Sumit",26,45000.0)
print("Before incrementing :")
print("_____________________________");
e1.display()
e2.display()
e1.increase_sal()
e2.increase_sal()
print()
print("After incrementing by",Emp.raise_amount,"percent:")
print("_____________________________________________");
e1.display()
e2.display()
```

Declaring Class Variable Inside Constructor

- We can declare a **class variable** inside the **constructor** also by **prefixing** the variable name with the **name of the class** and **dot** operator

- **Syntax:**

```
class <class_name>:

def __init__(self):
    <class name>.<var_name>=<value>
    self.<var_name>=<value>
    .
    .
    .
```

This is called a **class variable**

Example

```
class CompStudent:

  def __init__(self,name,roll):
    CompStudent.stream='cse'
    self.name = name
    self.roll = roll

obj1 = CompStudent('Atul',1)
obj2 = CompStudent('Chetan', 2)
print(obj1.name)
print(obj1.roll)
print(obj1.stream)
print(obj2.name)
print(obj2.roll)
print(obj2.stream)
print(CompStudent.stream)
```

We have shifted the var decl from class body to constructor body , but still it will be treated as class variable because we have prefixed it with classnname

Output:

```
Atul
1
cse
Chetan
2
cse
cse
```

Declaring Class Variable Inside Instance Method

- We can declare a **class variable** inside an instance method also also by **prefixing** the variable name with the **name of the class** and **dot** operator

- **Syntax:**

class <class_name>:

def <method_name>(self):

<class name>.<var_name>=<value>

self.<var_name>=<value>

.

.

.

This is called a class variable

Example

```
class Circle:
    def __init__(self,radius):
        self.radius=radius
    def cal_area(self):
        Circle.pi=3.14
        self.area=Circle.pi * self.radius ** 2
c1=Circle(10)
c2=Circle(20)
c1.cal_area()
print("radius=",c1.radius,"area=",c1.area,"pi=",Circle.pi)
c2.cal_area()
print("radius=",c2.radius,"area=",c2.area,"pi=",Circle.pi)
```

We have shifted the var decl from class body to method body , but still it will be treated as class variable because we have prefixed it with classnname

Output:

```
radius= 10 area= 314.0 pi= 3.14
radius= 20 area= 1256.0 pi= 3.14
```

Obtaining Details Of Class Variables

- As we know , **class variables** are owned by a class itself (i.e., by its definition), so to store their details a class also uses a dictionary called **__dict**

- Thus we can see that Python has **2 dictionaries** called **__dict__**.

- One is ***<class_name>.__dict__*** and the other is ***<object_ref>.__dict__***

Guess The Output ?

```
class Emp:
    raise_per=7.5
    comp_name="Google"
    def __init__(self):
            self.name="Amit"
            self.age=24
            self.sal=50000.0
e1=Emp()
print(e1.__dict__)
print()
print(Emp.__dict__)
```

Output:

```
{'name': 'Amit', 'age': 24, 'sal': 50000.0}

{'__module__': '__main__', 'raise_per': 7.5, 'comp_name': 'Google', '__init__':
<function Emp.__init__ at 0x00000000028179D8>, '__dict__': <attribute '__dict__'
 of 'Emp' objects>, '__weakref__': <attribute '__weakref__' of 'Emp' objects>, '
__doc__': None}
```

How many class variables will be created by this code?

python

```
class Sample:
    i=10
    def _init___(self):
        Sample.j=20
    def f1(self):
        Sample.k=30
Sample.m=40
print(Sample.__dict__)
```

Why the code is showing only **2 class variables** even though we have **4** ?

This is because the class variable **k** will only be created when **f1()** gets called . Similarly the variable **j** will be created when we will create any object of the class . But since **we didn't create any object** nor **we have called the method f1( )** so only **2 class variables** are there called **i** and **m**

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x0
000000022A79D8>, 'f1': <function Sample.f1 at 0x00000000022A7A60>, '__dict__':
<attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref
__' of 'Sample' objects>, '__doc__': None, 'm': 40}
```

How many class variables will be created by this code?

```
class Sample:
    i=10
    def __init___(self):
        Sample.j=20
    def f1(self):
        Sample.k=30
Sample.m=40
s1=Sample()
print(Sample.___dict___)
```

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x0
000000002DD79D8>, 'f1': <function Sample.f1 at 0x0000000002DD7A60>, '__dict__':
<attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref
__' of 'Sample' objects>, '__doc__': None, 'm': 40, 'j': 20}
```

Three class variables will be created by the code called i,j and m

How many class variables will be created by this code?

```
class Sample:
    i=10
    def __init__(self):
        Sample.j=20
    def f1(self):
        Sample.k=30
Sample.m=40
s1=Sample()
S2=Sample()
print(Sample.__dict__)
```

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x0
000000002DD79D8>, 'f1': <function Sample.f1 at 0x0000000002DD7A60>, '__dict__':
<attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref
__' of 'Sample' objects>, '__doc__': None, 'm': 40, 'j': 20}
```

Still only three **class variables** will be created by the code called **i,j** and **m** because **class variables** are not created **per instance basis** rather there is only **1 copy** shared by all the objects

How many class variables will be created by this code?

```
class Sample:
    i=10
    def __init__(self):
        Sample.j=20
    def f1(self):
        Sample.k=30
Sample.m=40
s1=Sample()
s2=Sample()
s1.f1()
s2.f1()
print(Sample.__dict__)
```

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x0
00000000029779D8>, 'f1': <function Sample.f1 at 0x000000002977A60>, '__dict__':
<attribute '__dict__' of 'Sample' objects>, '__weakref__': <attribute '__weakref
__' of 'Sample' objects>, '__doc__': None, 'm': 40, 'j': 20, 'k': 30}
```

Guess The Output ?

```
class Sample:
    i=10
    def __init__(self):
        print("Constructor called. . .")
        print(Sample.i)
        print(self.i)
    def f1(self):
        print("f1 called. . .")
        print(Sample.i)
        print(self.i)
s1=Sample()
s1.f1()
```

Output:

```
Constructor called. . .
10
10
f1 called. . .
10
10
```

Guess The Output ?

```
class Sample:
    i=10
    def __init__(self):
        self.i=20

s1=Sample()
print(Sample.i)
```

As mentioned previously , if we use **self** or **object reference** to **modify** a **class variable** , then **Python** does not **modify** the **class variable** . Rather it creates a new **instance variable** inside the **object's memory area** by the **same name.**

So in our case **2 variables** by the name **i** are created . **One** as **class variable** and **other** as **instance variable**

Output:

10

Guess The Output ?

```
class Sample:
   i=10
   def __init__(self):
        self.i=20


s1=Sample()
print(Sample.i)
print(s1.i)
```

Output:

```
10
20
```

Guess The Output ?

```
class Sample:
    i=10
    def __init__(self):
        Sample.i=20

s1=Sample()
print(Sample.i)
print(s1.i)
```

Output:

```
20
20
```

Guess The Output ?

```
class Sample:
    i=10
    def __init__(self):
        Sample.i=20

s1=Sample()
s1.i=30
print(Sample.i)
print(s1.i)
```

Output:

```
20
30
```

Guess The Output ?

```
class Sample:
    i=10
    def __init__(self):
          self.j=20
s1=Sample()
s2=Sample()
s1.i=100
s1.j=200
print(s1.i,s1.j)
print(s2.i,s2.j)
```

Output:

```
100 200
10 20
```

Guess The Output ?

```
class Sample:
   i=10
   def f1(self):
          self.j=20
s1=Sample()
s2=Sample()
s1.i=100
s1.j=200
print(s1.i,s1.j)
print(s2.i,s2.j)
```

Output:

```
100 200
Traceback (most recent call last):
  File "classdemo15.py", line 11, in <module>
    print(s2.i,s2.j)
AttributeError: 'Sample' object has no attribute 'j'
```

Deleting Class Variables

- We can **delete/remove** instance variables in 2 ways
  - Using **del classname .<var_name>** from anywhere in the program
  - Using **del cls.<var_name>** from **classmethod**
- **Special Note**: We cannot **delete** a **class variable** using **object reference** or **self** , otherwise **Python** will throw **AttributeError** exception

Guess The Output ?

```
class Sample:
    i=10
    def __init__(self):
        del Sample.i


print(Sample.__dict__)
s1=Sample()
print()
print(Sample.__dict__)
```

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x0
00000002A379D8>, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__wea
kref__': <attribute '__weakref__' of 'Sample' objects>, '__doc__': None}

{'__module__': '__main__', '__init__': <function Sample.__init__ at 0x0000000002
A379D8>, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__weakref__':
<attribute '__weakref__' of 'Sample' objects>, '__doc__': None}
```

Guess The Output ?

```
class Sample:
    i=10
    def __init__(self):
            del self.i


print(Sample.__dict__)
s1=Sample()
print()
print(Sample.__dict__)
```

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x0
000000002B079D8>, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__wea
kref__': <attribute '__weakref__' of 'Sample' objects>, '__doc__': None}
Traceback (most recent call last):
  File "classdemo15.py", line 7, in <module>
    s1=Sample()
  File "classdemo15.py", line 4, in __init__
    del self.i
AttributeError: i
```

Guess The Output ?

```
class Sample:
    i=10
    def __init__(self):
        del Sample.i

print(Sample.__dict__)
s1=Sample()
del  Sample.i
print()
print(Sample.__dict__)
```

Output:

```
{'__module__': '__main__', 'i': 10, '__init__': <function Sample.__init__ at 0x0
000000002DE79D8>, '__dict__': <attribute '__dict__' of 'Sample' objects>, '__wea
kref__': <attribute '__weakref__' of 'Sample' objects>, '__doc__': None}
Traceback (most recent call last):
  File "classdemo15.py", line 8, in <module>
    del Sample.i
AttributeError: i
```

PYTHON

LECTURE 41

Today's Agenda

- **Introduction To Object Oriented Programming-IV**
  - Class Methods
  - Creating Class Methods
  - Accessing Class Methods
  - Static Methods
  - Accessing Static Methods
  - Difference Between Instance Method , Class Method and Static Methods

Class Methods

- Just like we can have **class variables** , similarly **Python** also allows us to create **class methods**.

- These are those methods ***which work on the class as a whole*** , instead of working on it's **object**.

- For , example in our **Emp** class if we want to initialize the class variable **raise_per** inside a method , then the best way would be to create a **class method** for this purpose

Creating A Class Metod

- To create a **class method** we write the special word **@classmethod** on top of method definition

- **Syntax:**

```
class <class_name>:
@classmethod
def  <method_name>(cls):
   // class specific code
```

This is called decorator

Notice that a class method gets a special object reference passed as argument by Python called as class refercnce

Important Points About ClassMethods

- To define a **class method** it is compulsory to use the decorator **@classmethod**
- **ClassMethods** can only access **class level data** and not **instance specific data**

Important Points About ClassMethods

- Just like **Python** passed **self** as argument to **instance methods** , it automatically passes **cls** as argument to **classmethods**

- The argument **cls** is always passed as the first argument and represents the **class object**.

Important Points About ClassMethods

- Recall , that for every class **Python** creates a special object called class object , so the reference **cls** points to this object.

- The name **cls** is just a convention , although we can give any name to it.

Important Points About ClassMethods

- To call a **classmethod** we simply prefix it with **classname** followed by dot operator.

- Although we can use **object reference** also to call a **classmethod** but ***it is highly recommended not to do so*** , since **classmethods** do not work upon **individual instances** of the class

Exercise

- Write a program to create a class called **Emp** , having an **instance members** called **name** , **age** and **sal** . Also declare a **class variable** called **raise_amount** to store the **increment percentage** of **sal** **and set it the value given by the user**
- Now provide following methods in your class
  - **__init__()** : This method should initialize instance members with the parameter passed
  - **increase_sal():** This method should calculate the increment in sal and add ot to the instance member sal
  - **display()**: This method should display name , age and sal of the employee
- Finally , in the main script , **create 2 Emp objects** , **initialize them** and **increase their salary** . Finally **display** the data

Output:

```
Enter raise percentage:8.5
Before incrementing :
________________________________________
Amit 24 50000.0
Sumit 26 45000.0

After incrementing by 8.5 percent:
________________________________________
Amit 24 54250.0
Sumit 26 48825.0
```

PYTHON

LECTURE 42

Today's Agenda

- **Advance Concepts Of Object Oriented Programming-I**
  - Encapsulation
  - Does Python Support Encapsulation ?
  - How To Declare Private Members In Python ?
  - The object Class And The __str__() Method
  - The Destructor

Encapsulation

- **Encapsulation** is the packing of ***data and functions operating on that data*** into a **single component** and ***restricting the access to some of the object's components.***

- **Encapsulation** means that the internal representation of an object is **generally hidden** from view **outside of the class body.**

Is The Following Code Supporting Encapsulation ?

```
class Emp:
  def __init__(self):
        self.age=25
        self.name="Rahul"
        self.salary=30000.0

e=Emp()
print("Age:",e.age,"Name:",e.name,"Salary:",e.salary)
```

No , the following code is violating **Encapsulation** as it is allowing us to **access data members** from **outside the class** directly using object

Output:

```
Age: 25 Name: Rahul Salary: 30000.0
```

How To Achieve Encapsulation In Python ?

- To achieve **Encapsulation** in **Python** we have to prefix the data member name with **double underscore**

- **Syntax:**

self.__<var_name>=<value>

Achieving Encapsulation

```
class Emp:
    def __init__(self):
        self.age=25
        self.name="Rahul"
        self.__salary=30000.0

e=Emp()
print("Age:",e.age)
print("Name:",e.name)
print("Salary:",e.__salary)
```

Since we have created the data member as __salary so it has become a private member and cannot be accessed outside the class directly

Output:

```
Age: 25
Name: Rahul
Traceback (most recent call last):
  File "classdemo22.py", line 10, in <module>
    print("Salary:",e.__salary)
AttributeError: 'Emp' object has no attribute '__salary'
```

Achieving Encapsulation

- Now to access such **private members** , we must define **instance methods** in the class

- From **outside the class** we must call these **methods** using **object** instead of directly accessing **data members**

Achieving Encapsulation

```
class Emp:
    def __init__(self):
            self.__age=25
            self.__name="Rahul"
            self.__salary=30000.0
    def show(self):
            print("Age:",self.__age,"Name:",self.__name,"Salary:",self.__salary)

e=Emp()
e.show()
```

Output:

```
Age: 25 Name: Rahul Salary: 30000.0
```

Private Methods

- Just like we have **private data members** , we also can have **private methods** .

- The syntax is also same.

- Simply **prefix** the **method name** with **double underscore** to make it a **private method**

Private Methods

```
class Emp:
    def __init__(self):
        self.__age=25
        self.__name="Rahul"
        self.__salary=30000.0
    def __show(self):
        print("Age:",self.__age,"Name:",self.__name,"Salary:",self.__salary)

e=Emp()
e.__show()
```

Output:

```
Traceback (most recent call last):
  File "classdemo22.py", line 10, in <module>
    e.__show()
AttributeError: 'Emp' object has no attribute '__show'
```

An Important Point

- When we declare a data member with double underscore indicating that it is private , **Python** actually **masks** it

- In other words , **Python** *changes the name of the variable* by using the syntax **_<classname>__<attributename>**

- **For example** , **__age** will actually become **_Emp__age**

So, What It Means To Us ?

- This means that **private attributes** are **not actually private** and are not prevented by **Python** from getting accessed from outside the class.

- So if they are **accessed** using the **above mentioned syntax** then no **Error** or **Exception** will arise

- So , finally we can say **NOTHING IN PYTHON IS ACTUALLY PRIVATE**

Accessing Private Data

```
class Emp:
    def __init__(self):
            self.__age=25
            self.__name="Rahul"
            self.__salary=30000.0
    def show(self):
            print("Age:",self.__age,"Name:",self.__name,"Salary:",self.__salary)

e=Emp()
e.show()
print("Age:",e._Emp__age,"Name:",e._Emp__name,"Salary:",e._Emp__salary)
```

Output:

```
Age: 25 Name: Rahul Salary: 30000.0
Age: 25 Name: Rahul Salary: 30000.0
```

The __str__() Method

- In **Python** , whenever we try to print an **object reference** by passing it's name to the **print()** function , we get **2 types** of **outputs**:
  - For **predefined classes** like **list** ,**tuple** or **str** , we get the **contents of the object**
  - For **our own class objects** we get the **class name** and the **id** of the **object instance** (which is the object's memory address in **CPython**.)

Why Is It So ?

- This is because **whenever we pass an object reference name to the print() function** , **Python internally calls a special instance method** available in **our class.**

- This method is called **__str__()** .

From where this method came ?

- From **Python 3.0** onwards , every class which we create always automatically inherits from the class **object**

- Or , we can say that **Python** implicitly inherits our class from the class **object**.

- The class **object** defines some special methods which every class inherits .

- Amongst these special methods some very important are **__init__()**, **__str__()**,**__new__()** etc

Can we see all the members of object class ?

- **Yes , it is very simple!**

- Just create an instance of **object** class and call the function **dir( )** .

- Recall that we used **dir( )** to print names of all the **members** of a **module** .

- Similarly we also can use **dir( )** to **print names** of all the members of any class by passing it the i**nstance** of the class as **argument**

Example

obj=object()
print(type(obj))
print(dir(obj))

Output:

```
<class 'object'>
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge
__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr_
_', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

The __str__() Method

- Now , if we do not redefine (override) this method in our class , then **Python** calls it's **default implementation** given by **object class** which is designed in such a way that it **returns** the **class name** followed by **object's memory address**

- However all built in classes like **list** , **str** , **tuple** , **int** , **float** , **bool** etc have **overridden** this method in such a way that it returns the content of the object.

Overriding __str__()

- So if we also want the same behaviour for our object then we also can **override** this method in our class in such a way that it returns the **content of the object**.

- The only point we have to remember while **overriding** this method is that ***it should return a string value***

Overriding __str__()

```
class Emp:
    def __init__(self,age,name,salary):
        self.age=age
        self.name=name
        self.salary=salary
    def __str__(self):
        return f"Age:{self.age},Name:{self.name},Salary:{self.salary}"

e=Emp(25,"Rahul",30000.0)
print(e)
```

Output:

```
Age:25,Name:Rahul,Salary:30000.0
```

Destructor

- Just like a **constructor** is used to **initialize** an object, a **destructor** is used to destroy the object and perform the final clean up.

- But a question arises that if we already have **garbage collector** in **Python** to clean up the memory , then ***why we need a destructor ?***

Destructor

- Although in python we do have **garbage collector** to **clean up the memory**, but it's not just memory which has to be freed when an object is dereferenced or destroyed.

- There can be a **lot of other resources as well**, like **closing open files**, **closing database connections** etc.

- Hence when we might require a **destructor** in our class for this purpose

Destructor In Python

- Just like we have **__init__()** which can be considered like a constructor as it initializes the object , similarly in **Python** we have another magic method called **__del__().**

- This method is automatically called by **Python** whenever an **object reference** goes **out of scope** and the **object** is **destroyed**.

Guess The Output ?

```
class Test:
    def __init__(self):
        print("Object created")

    def __del__(self):
        print("Object destroyed")
t=Test()
```

Output:

```
Object created
Object destroyed
```

Since at the end of the code , Python collects the object through it's garbage collector so it automatically calls the __del__() method also

How To Force Python To Call __del__() ?

- If we want to force **Python** to call the **__del__()** method , then we will have to forcibly destroy the object

- To do this we have to use **del operator** passing it the **object reference**

Guess The Output ?

```
class Test:
    def __init__(self):
        print("Object created")

    def __del__(self):
        print("Object destroyed")
t1=Test()
del t1
print("done")
```

Output:

```
Object created
Object destroyed
done
```

Guess The Output ?

python

```
class Test:
  def __init__(self):
        print("Object created")

  def __del__(self):
        print("Object destroyed")
t1=Test()
t2=t1
del t1
print("done")
```

Output:

```
Object created
done
Object destroyed
```

We must remember that Python destroys the object only when the reference count becomes 0 . Now in this case after deleting t1 , still the object is being refered by t2 . So the __del__() was not called on del t1. It only gets called when t2 also goes out of scope at the end of the program and reference count of the object becomes 0

Guess The Output ?

```
class Test:
    def __init__(self):
        print("Object created")

    def __del__(self):
        print("Object destroyed")
t1=Test()
t2=t1
del t1
print("t1 deleted")
del t2
print("t2 deleted")
print("done")
```

Output:

```
Object created
t1 deleted
Object destroyed
t2 deleted
done
```

PYTHON

LECTURE 43

Today's Agenda

- **Advance Concepts Of Object Oriented Programming-II**
  - Inheritance
  - Types Of Inheritance
  - Single Inheritance
  - Using super( )
  - Method Overriding

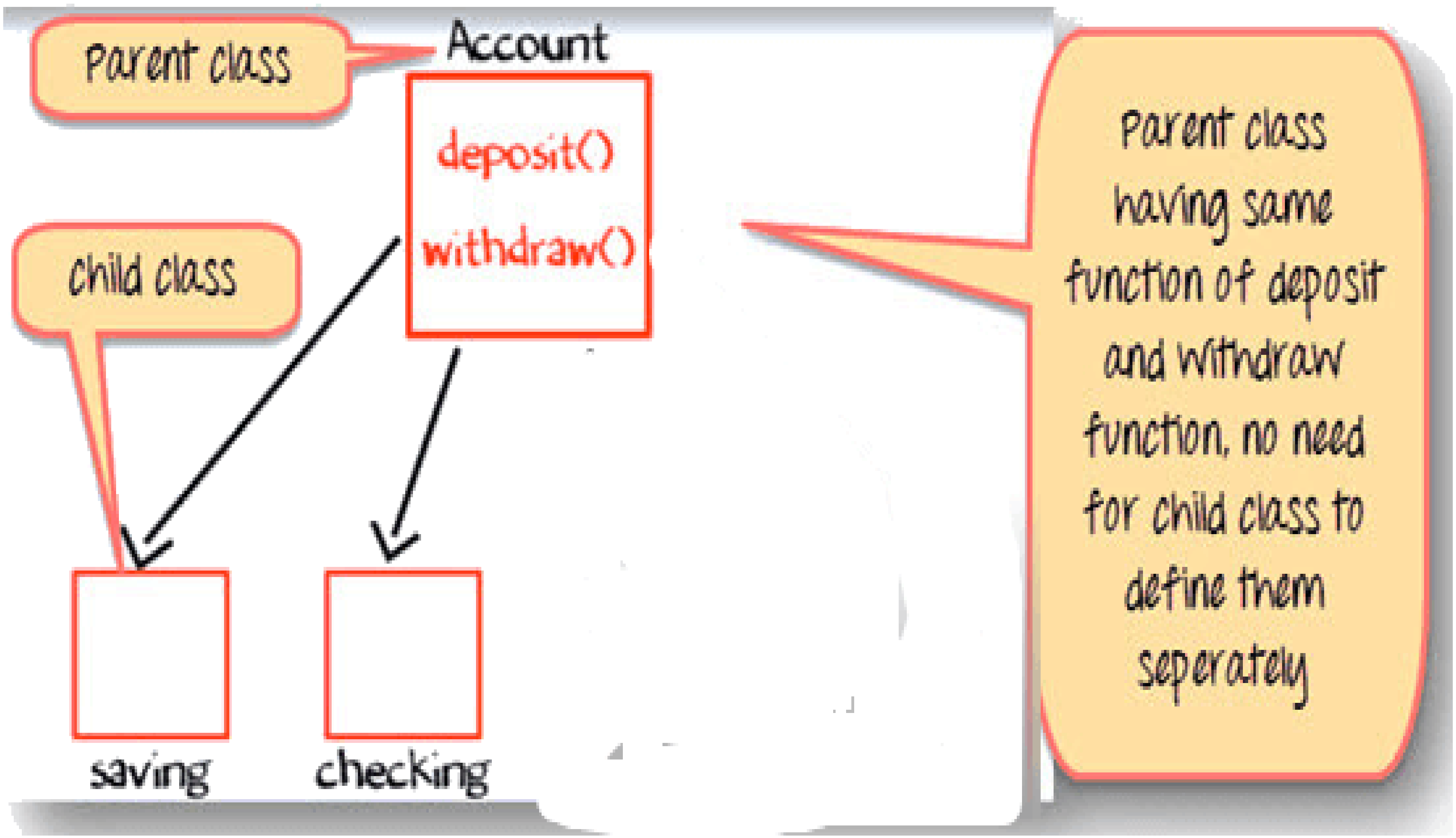
Inheritance

- Inheritance is a **powerful feature** in **object oriented programming.**

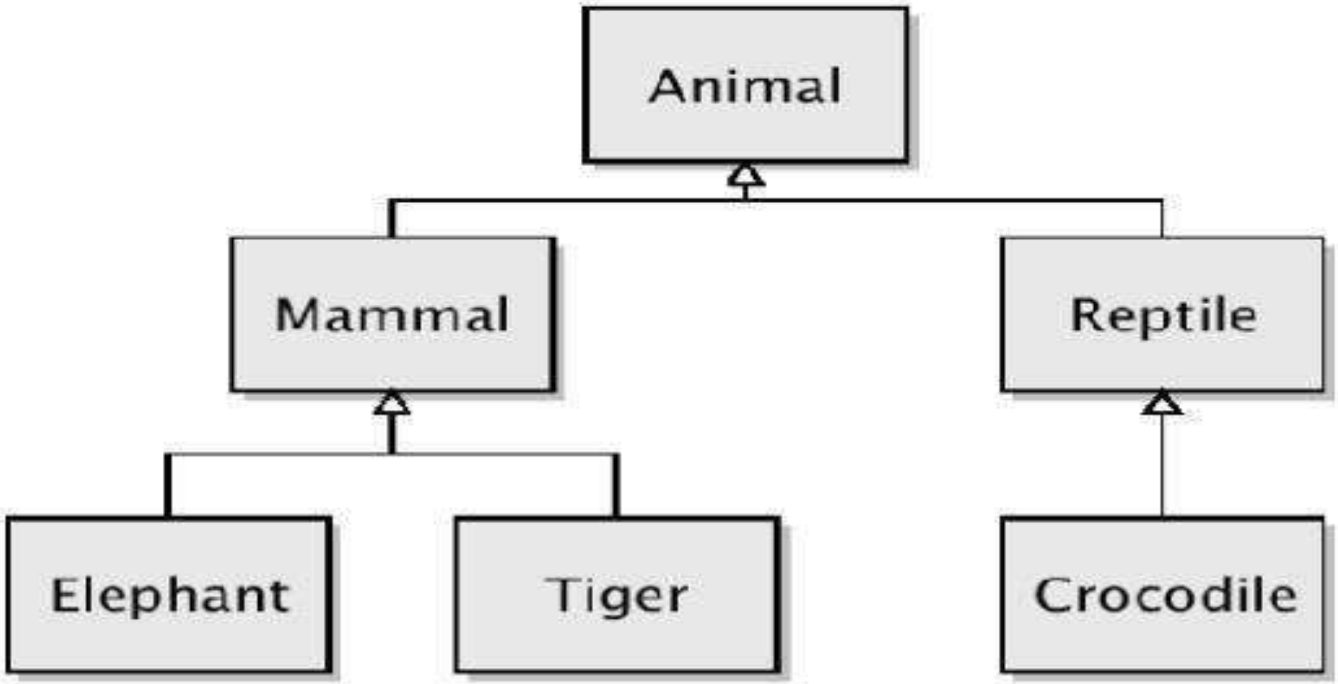
- It refers to defining a **new class** with **little or no modification** to an **existing class**.

- The **new class** is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

Real Life Examples



Real Life Examples



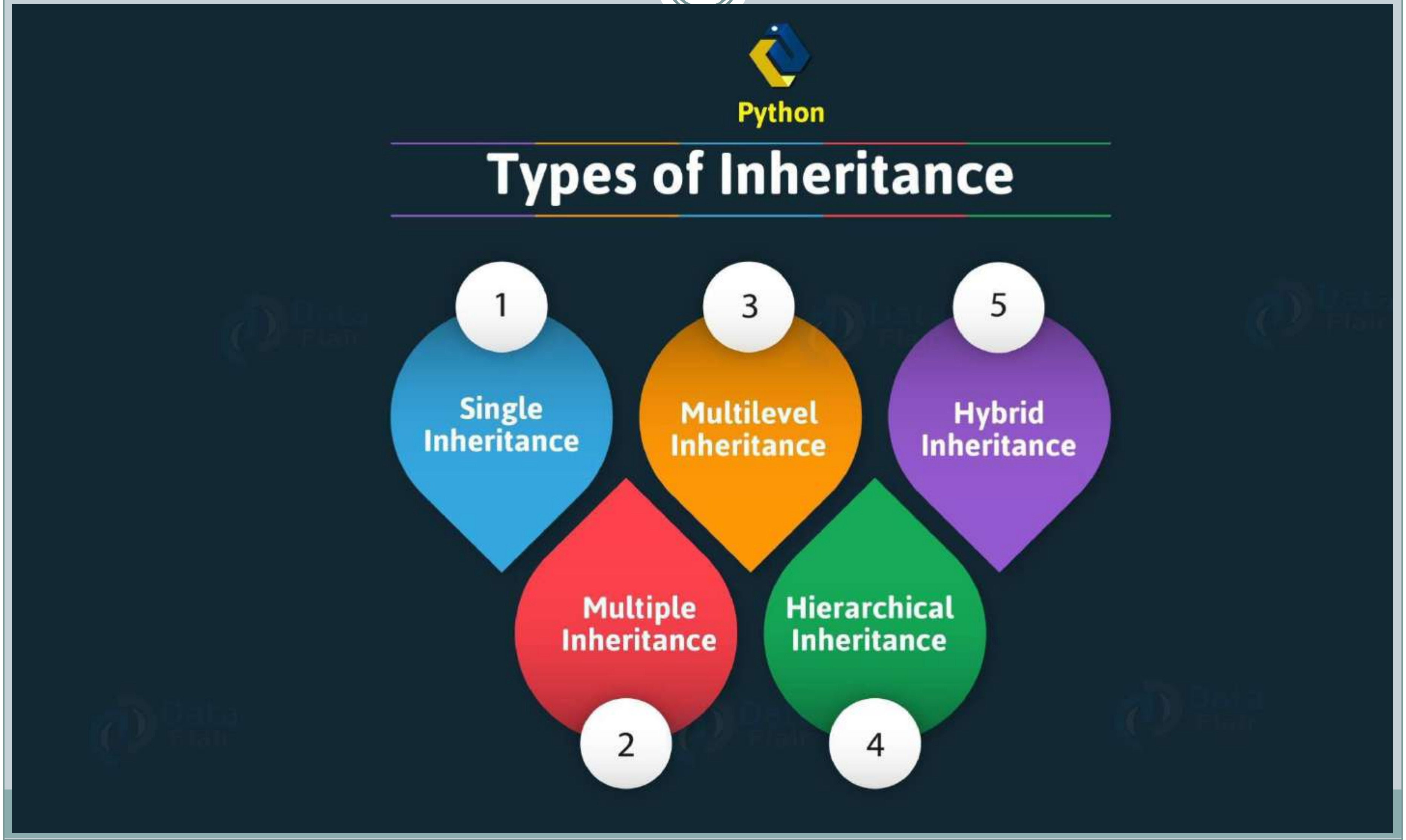
Benefits

- It represents **real-world relationships** well.

- It provides **reusability** of a code. We **don’t have to** write the **same code again and again**.

- It also allows us to **add more features** to a class without modifying it.

Types Of Inheritance Supported By Python



Syntax Of Single Inheritance In Python

```
class BaseClass:
    Body of base class

class DerivedClass(BaseClass):
    Body of derived class
```

For Ex:

```
class Account:
    pass
class SavingAccount(Account):
    pass
```

Example

```
class Animal:
    def eat(self):
        print("It eats.")
    def sleep(self):
        print("It sleeps.")

class Bird(Animal):
    def set_type(self,type):
        self.type=type
    def fly(self):
        print("It flies in the sky.")
    def __str__(self):
        return "This is a "+self.type;
```

```
duck=Bird()
duck.set_type("Duck")
print(duck)
duck.eat()
duck.sleep()
duck.fly()
```

Output:

```
This is a Duck
It eats.
It sleeps.
It flies in the sky.
```

Using super()

- In Python , to call parent class members from the child class we can use the method **super( ).**

- Using **super()** is required in 2 situations:

  - **For calling parent class constructor**

  - **For calling overridden methods**

How Constructors Behave In Inheritance ?

- Whenever we create a **child class object** , **Python** looks for **__init__()** method in **child class**.

- If the **child class** doesn't contain an _ **init__()** method then **Python** goes up in the inheritance chain and looks for the **__init__()** method of **parent class**

How Constructors Behave In Inheritance ?

- If the parent class contains ___**init**___**()** , then it executes it .

- **Now an important point to notice** is that if child class also has ___**init**___**()** , then **Python** will not call parent's **___init___()** method.

- That is , unlike **Java** or **C++** , **Python** does not automatically call the parent class ___**init**___**()** if it finds an **___init___()** method in **child class**

How Constructors Behave In Inheritance ?

```
class A:
  def __init__(self):
      print("Instantiating A...")

class B(A):
    pass

b=B()
```

Output:

```
Instantiating A...
```

As you can see, Python called the constructor of class A , since B class doesn’t has any constructor defined

How Constructors Behave In Inheritance ?

```
class A:
    def __init__(self):
        print("Instantiating A...")


class B(A):
    def __init__(self):
        print("Instantiating B...")


b=B()
```

This time , Python did not call the constructor of class A as it found a constructor in B itself

Output:

```
Instantiating B...
```

How Constructors Behave In Inheritance ?

- **So , what is the problem if parent constructor doesn’t get called ?**

- The problem is that , if parent class **constructor doesn’t get** called then all the **instance members it creates** will **not be made available to child class**

How Constructors Behave In Inheritance ?

```
class Rectangle:
    def __init__(self):
            self.l=10
            self.b=20
class Cuboid(Rectangle):
    def __init__(self):
            self.h=30
    def volume(self):
            print("Vol of cuboid is",self.l*self.b*self.h)
obj=Cuboid()
obj.volume()
```

Since , constructor of **Rectangle** was not called , so the expression **self.l** produced exception because there is no **attribute** created by the name of **l**

Output:

```
Traceback (most recent call last):
  File "inhdemo2.py", line 15, in <module>
    obj.volume()
  File "inhdemo2.py", line 10, in volume
    print("Vol of cuboid is",self.l*self.b*self.h)
AttributeError: 'Cuboid' object has no attribute 'l'
```

How Can We Explicitly Call __init__() Of Parent Class ?

- If we want to call the parent class __init__() , then we will have 2 options:

  - **Call it using the name of parent class explicitly**
  - **Call it using the method super()**

Calling Parent Constructor Using Name

```
class Rectangle:
    def __init__(self):
        self.l=10
        self.b=20

class Cuboid(Rectangle):
    def __init__(self):
        Rectangle.__init__(self)
        self.h=30
    def volume(self):
        print("Vol of cuboid is",self.l*self.b*self.h)

obj=Cuboid()
obj.volume()
```

Notice that we have to explicitly pass the argument self while calling __init__() method of parent class

Output:

```
Vol of cuboid is 6000
```

Calling Parent Constructor Using super( )

```
class Rectangle:
    def __init__(self):
        self.l=10
        self.b=20

class Cuboid(Rectangle):
    def __init__(self):
        super().__init__();
        self.h=30
    def volume(self):
        print("Vol of cuboid is",self.l*self.b*self.h)

obj=Cuboid()
obj.volume()
```

Again notice that this time we don't have to pass the argument **self** when we are using **super( )** as Python will automatically pass it

Output:

```
Vol of cuboid is 6000
```

What Really Is **super( )** ?

- The method **super()** is a **special method** made available by **Python** which returns *a* ***proxy object*** *that delegates method calls to a* ***parent class***

- In simple words the method **super( )** provides us a special object that can be used to transfer call to parent class methods

Benefits Of super( )

- A common question that arises in our mind is that why to use **super( ) ,** if we can call the parent class methods using **parent class name.**

- The answer is that **super( )** gives **4 benefits**:

  - We don't have to pass **self** while calling any method using **super( ).**
  - If the **name of parent class changes** after inheritance then we will not have to rewrite the code in child **as super( ) will automatically connect itself to current parent**
  - It can be used to resolve **method overriding**
  - It is very helpful in **multiple inheritance**

Method Overriding

- To understand **Method Overriding** , try to figure out the output of the code given in the next slide

Guess The Output ?

```
class Person:
    def __init__(self,age,name):
            self.age=age
            self.name=name
    def __str__(self):
            return f"Age:{self.age},Name:{self.name}"
class Emp(Person):
    def __init__(self,age,name,id,sal):
            super().__init__(age,name)
            self.id=id
            self.sal=sal

e=Emp(24,"Nitin",101,45000)
print(e)
```

Output:

```
Age:24,Name:Nitin
```

Explanation

- As we know , whenever we **pass** the name of an **object reference** as **argument** to the function **print( )** , **Python** calls the method __**str__().**

- But since the class **Emp** doesn't has this method , so **Python moves up in the inheritance chain** to find this method in the base class **Person**

- Now since the class **Person** has this method , **Python** calls the __**str__()** method of **Person** which returns only the **name** and **age**

Method Overriding

- Now if we want to change this **behavior** and show all **4 attributes** of the Employee i.e. his **name** , **age** ,**id** and **salary**, then we will have to **redefine the method __str__()** in our **Emp** class.

- This is called **Method Overriding**

- Thus , **Method Overriding** is a concept in **OOP** which occurs whenever a **derived class redefines** the **same method** as **inherited** from the **base class**

Modified Example

```
class Person:
    def __init__(self,age,name):
        self.age=age
        self.name=name
    def __str__(self):
        return f"Age:{self.age},Name:{self.name}"
class Emp(Person):
    def __init__(self,age,name,id,sal):
        super().__init__(age,name)
        self.id=id
        self.sal=sal
    def __str__(self):
        return f"Age:{self.age},Name:{self.name},Id:{self.id},Salary:{self.sal}"
e=Emp(24,"Nitin",101,45000)
print(e)
```

Output:

```
Age:24,Name:Nitin,Id:101,Salary:45000
```

Role Of **super( )** In Method Overriding

- When we **override** a method of **base class** in the **derived class** then **Python** will always call the **derive's version** of the method.

- But in some cases we also want to call the **base class version** of the **overridden** method.

- In this case we can call the **base class version** of the method from the **derive class** using the function **super( )**
- **Syntax:**

  super( ). <method_name>(<arg>)

Modified Example

```
class Person:
    def __init__(self,age,name):
            self.age=age
            self.name=name
    def __str__(self):
            return f"Age:{self.age},Name:{self.name}"
class Emp(Person):
    def __init__(self,age,name,id,sal):
            super().__init__(age,name)
            self.id=id
            self.sal=sal
    def __str__(self):
            str=super().__str__()
            return f"{str},Id:{self.id},Salary:{self.sal}"
e=Emp(24,"Nitin",101,45000)
print(e)
```

<u>Output:</u>

```
Age:24,Name:Nitin,Id:101,Salary:45000
```

Exercise

- Write a program to create a class called **Circle** having an instance member called **radius**. Provide following methods in **Circle** class
  - **__init___()** : This method should accept an argument and initialize radius with it
  - **area():** This method should calculate and return Circle's area
- Now create a derived class of **Circle** called **Cylinder** having an instance member called **height.** Provide following methods in **Cylinder** class
  - **__init___()** : This method should initialize instance members **radius** and **height** with the parameter passed.
  - **area( ):** This method should override Circle's area( ) to calculate and return area of Cylinder . (formula: $2\pi r^2+2\pi rh$)
  - **volume():** This method should calculate and return Cylinder's volume(formula: $\pi r^2 h$)

Solution

```
import math
class Circle:
    def __init__(self,radius):
        self.radius=radius
    def area(self):
        return math.pi*math.pow(self.radius,2)
class Cylinder(Circle):
    def __init__(self,radius,height):
        super().__init__(radius)
        self.height=height
    def area(self):
        return 2*super().area()+2*math.pi*self.radius*self.height
    def volume(self):
        return super().area()*self.height
```

```
obj=Cylinder(10,20)
print("Area of cylinder is",obj.area())
print("Volume of cylinder is",obj.volume())
```

Output:

```
Area of cylinder is 1884.9555921538758
Volume of cylinder is 6283.185307179587
```

A Very Important Point!

- **Can we call the base class version of an overridden method from outside the derived class ?**

- **For example** , in the previous code we want to call the method **area( )** of **Circle** class from our **main script** . **How can we do this ?**

- Yes this is possible and for this **Python** provides us a special syntax:

- **Syntax:**

  <base_class_name>.<method_name>(<der_obj>)

Example

```
import math
class Circle:
    def __init__(self,radius):
        self.radius=radius
    def area(self):
        return math.pi*math.pow(self.radius,2)
class Cylinder(Circle):
    def __init__(self,radius,height):
        super().__init__(radius)
        self.height=height
    def area(self):
        return 2*super().area()+2*math.pi*self.radius*self.height
    def volume(self):
        return super().area()*self.height
```

```
obj=Cylinder(10,20)
print("Area of cylinder is",obj.area())
print("Volume of cylinder is",obj.volume())
print("Area of Circle:",Circle.area(obj))
```

By calling in this way we can bypass the area( ) method of Cylinder and directly call area( ) method of Circle

Output:

```
Area of cylinder is 1884.9555921538758
Volume of cylinder is 6283.185307179587
Area of Circle: 314.1592653589793
```

PYTHON

LECTURE 44

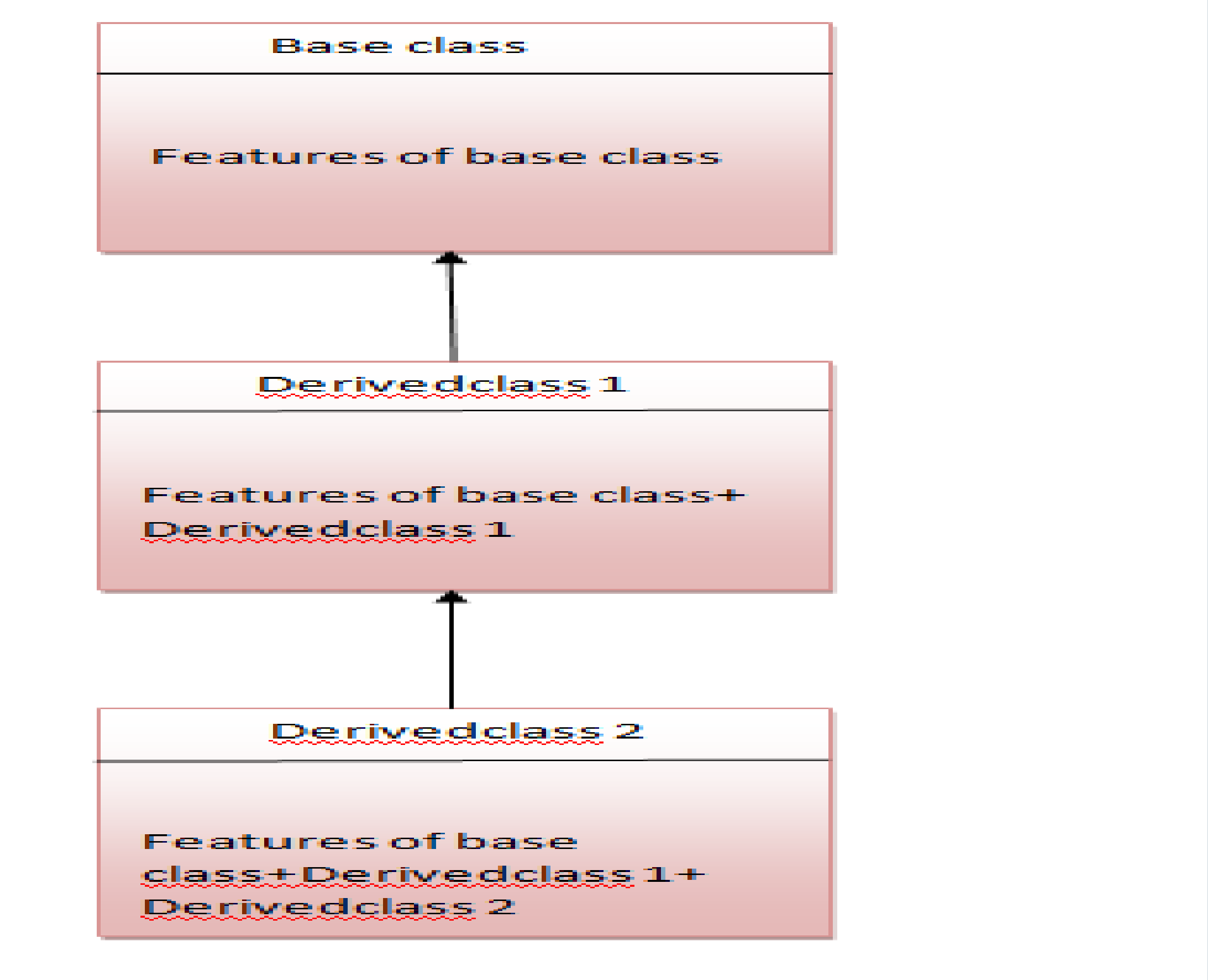
Today's Agenda

- **Advance Concepts Of Object Oriented Programming-III**
  - MultiLevel Inheritance
  - Hierarchical Inheritance
  - Using The Function issubclass( )
  - Using The Function isinstance( )

MultiLevel Inheritance

- **Multilevel inheritance** is also possible in Python like other Object Oriented programming languages.
- We can inherit a **derived class** from **another derived class.**
- This process is known as **multilevel inheritance.**
- In Python, **multilevel inheritance** can be done at any depth.

MultiLevel Inheritance

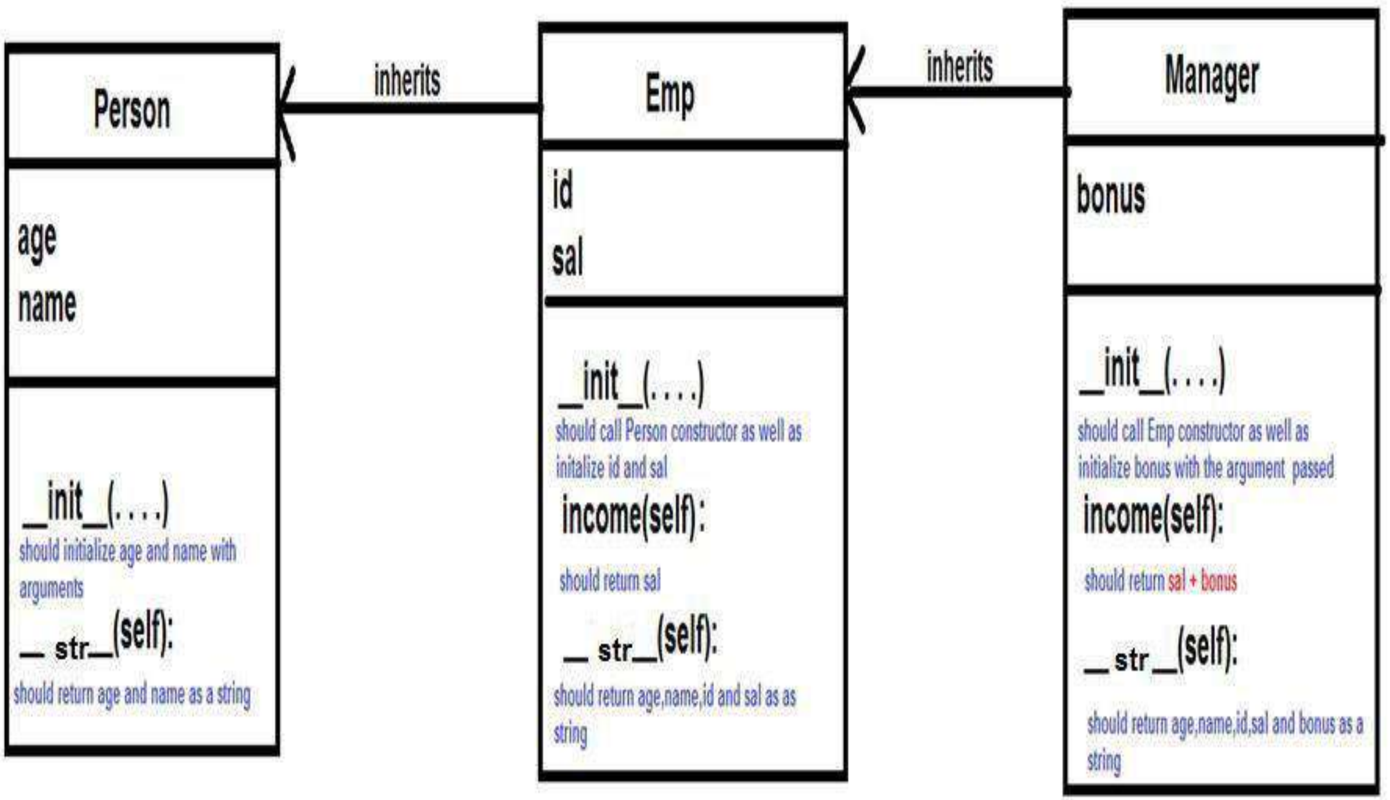


Syntax

```
class A:
    # properties of class A

class B(A):
    # class B inheriting property of class A
    # more properties of class B

class C(B):
    # class C inheriting property of class B
    # thus, class C also inherits properties of class A
    # more properties of class C
```

Exercise

- Write a program to create 3 classes **Person** , **Emp** and **Manager**.



Now in the main script create an instance of Manager class and initialize it with required values . Now display 3 things:
1. Complete details of Manager 2. Only the salary of Manager 3. Total income of Manager

Desired Output

```
Person constructor called. . .
Emp constructor called. . .
Manager constructor called. . .
Age:24,Name:Nitin,Id:101,Salary:45000,Bonus:20000
Manager's Salary: 45000
Manager's Total Income: 65000
```

Solution

```
class Person:
    def __init__(self,age,name):
            self.age=age
            self.name=name
            print("Person constructor called. . .")
    def __str__(self):
            return f"Age:{self.age},Name:{self.name}"
class Emp(Person):
    def __init__(self,age,name,id,sal):
            super().__init__(age,name)
            self.id=id
            self.sal=sal
            print("Emp constructor called. . .")
    def income(self):
            return self.sal

    def __str__(self):
            str=super().__str__()
            return f"{str},Id:{self.id},Salary:{self.sal}"
```

Solution

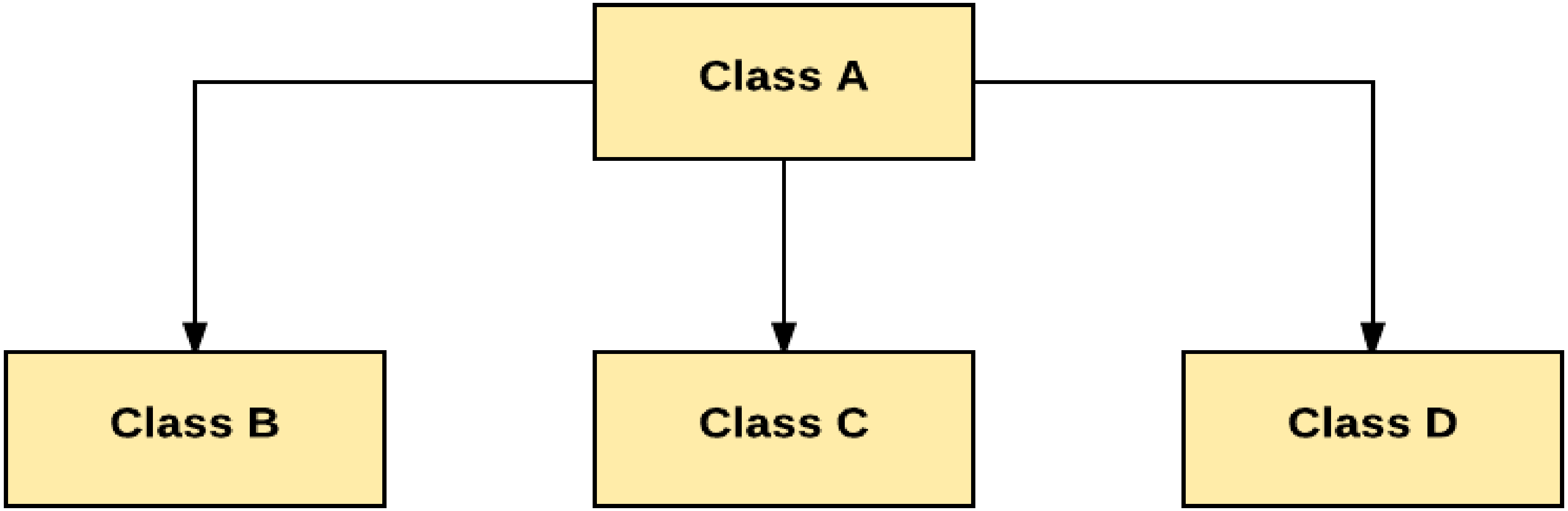
```
class Manager(Emp):
    def __init__(self,age,name,id,sal,bonus):
        super().__init__(age,name,id,sal)
        self.bonus=bonus
        print("Manager constructor called. . .")
    def income(self):
        total=super().income()+self.bonus
        return total
    def __str__(self):
        str=super().__str__()
        return f"{str},Bonus:{self.bonus}"


m=Manager(24,"Nitin",101,45000,20000)
print(m)
print("Manager's Salary:",Emp.income(m))
print("Manager's Total Income:",m.income())
```

Output:

```
Person constructor called. . .
Emp constructor called. . .
Manager constructor called. . .
Age:24,Name:Nitin,Id:101,Salary:45000,Bonus:20000
Manager's Salary: 45000
Manager's Total Income: 65000
```

Hierarchical Inheritance

- In **Hierarchical Inheritance**, **one class** is inherited by many **sub classes**.



Hierarchical Inheritance

- Suppose you want to write a program which has to keep track of the **teachers** and **students** in a college.

- They have **some common characteristics** such as **name** and **age**.

- They also have specific characteristics such as **salary** for **teachers** and **marks** for **students**.

Hierarchical Inheritance

- **One way** to solve the problem is that we can create **two independent classes** for **each type** and **process them**.

- But adding a **new common characteristic** would mean **adding to both** of these independent classes.

- This quickly becomes **very exhaustive task**

Hierarchical Inheritance

- A much better way would be to create a common class called **SchoolMember** and then have the **Teacher** and **Student** classes ***inherit*** from this class

- That is , they will become sub-types of this type (class) and then we can add specific characteristics to these sub-types

Example

```
class SchoolMember:

    def __init__(self, name, age):
        self.name = name
        self.age = age
        print("Initialized SchoolMember:",self.name)

    def tell(self):

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

Example

```
class Teacher(SchoolMember):

    def __init__(self, name, age, salary):
        super().__init__(name, age)
        self.salary = salary
        print("Initialized Teacher:", self.name)

    def tell(self):
        super().tell()
        print("Salary:",self.salary)
```

Example

```python
class Student(SchoolMember):

    def __init__(self, name, age, marks):
        super().__init__(name, age)
        self.marks = marks
        print("Initialized Student:",self.name)

    def tell(self):
        super().tell()
        print("Marks:",self.marks)
t = Teacher('Mr. Kumar', 40, 80000)
s = Student('Sudhir', 25, 75)
print()
members = [t, s]
for member in members:
      member.tell()
```

Output

```
Initialized SchoolMember: Mr. Kumar
Initialized Teacher: Mr. Kumar
Initialized SchoolMember: Sudhir
Initialized Student: Sudhir

Name: Mr. Kumar Age: 40 Salary: 80000
Name: Sudhir Age: 25 Marks: 75
```

How To Check Whether A Class Is A SubClass Of Another ?

- **Python** provides a function **issubclass()** that directly tells us if a class is a **subclass** of **another class.**

- **Syntax:**

issubclass(<name of der class>,<name of base class>)

- The **function** returns **True** if the **classname** passed as **first argument** is the derive class of the **classname** passed as **second argument** otherwise it returns **False**

Guess The Output ?

```
class MyBase(object):
    pass

class MyDerived(MyBase):
    pass

print(issubclass(MyDerived, MyBase))
print(issubclass(MyBase, object))
print(issubclass(MyDerived, object))
print(issubclass(MyBase, MyDerived))
```

Output:

```
True
True
True
False
```

Guess The Output ?

```
class MyBase:
    pass

class MyDerived(MyBase):
    pass

print(issubclass(MyDerived, MyBase))
print(issubclass(MyBase, object))
print(issubclass(MyDerived, object))
print(issubclass(MyBase, MyDerived))
```

Output:

```
True
True
True
False
```

In Python 3 , every class implicitly inherits from object class but in Python 2 it is not so. Thus in Python 2 the 2nd and 3rd print( ) statements would return False

Alternate Way

- Another way to do the same task is to call the function **isinstance( )**

- **Syntax:**

isinstance(<name of obj ref>,<name of class>)

- The **function** returns **True** if the **object reference** passed as **first argument** is an instance of the **classname** passed as **second argument** or any of it's **subclasses.** Otherwise it returns **False**

Guess The Output ?

```
class MyBase:
   pass

class MyDerived(MyBase):
   pass

d = MyDerived()
b = MyBase() print(isinstance(d,
MyBase)) print(isinstance(d,
MyDerived))
print(isinstance(d, object))
print(isinstance(b, MyBase))
print(isinstance(b, MyDerived))
print(isinstance(b, object))
```

Output:

```
True
True
True
True
False
True
```

PYTHON

LECTURE 45

Today's Agenda

- **Advance Concepts Of Object Oriented Programming-IV**
  - Multiple Inheritance
  - The MRO Algorithm
  - Hybrid Inheritance
  - The Diamond Problem

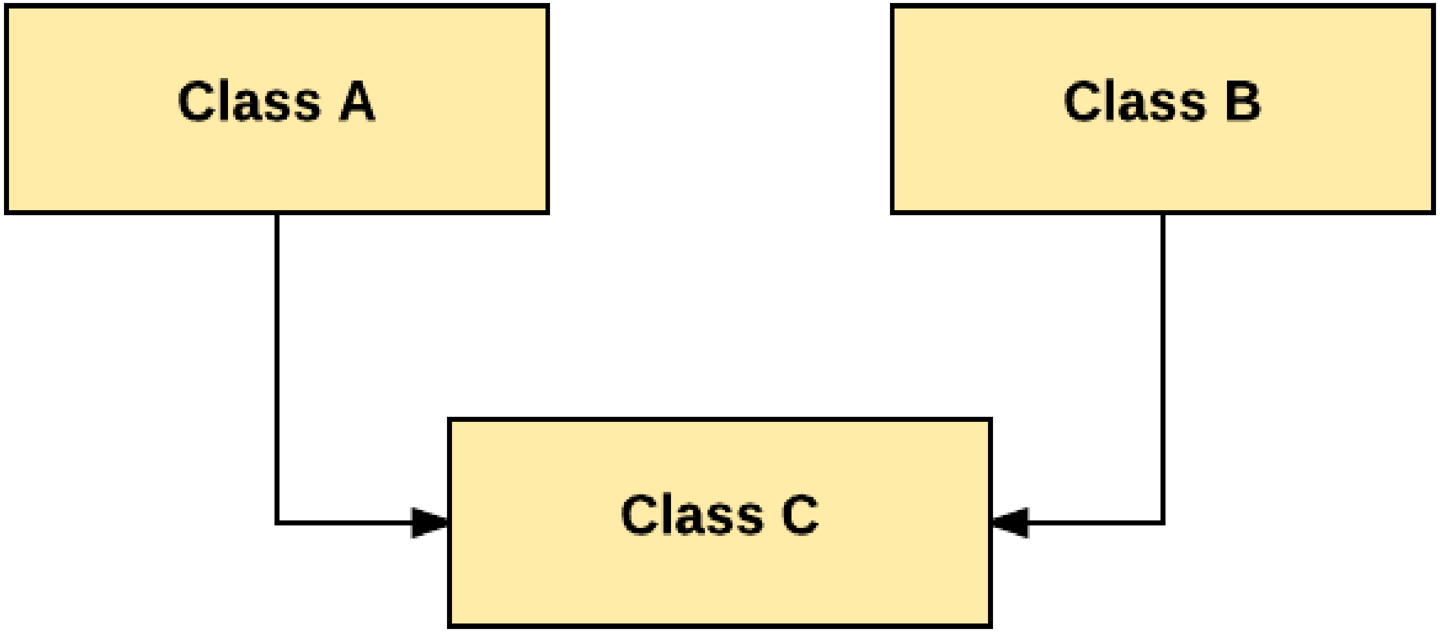
Multiple Inheritance

- Like **C++**, in **Python** also a class can be derived from more than one base class.

- This is called **multiple inheritance**.

- In **multiple inheritance**, the features of all the base classes are inherited into the derived class.

Multiple Inheritance



Syntax

```
class A:
    # properties of class A

class B:
    #properties of class B

class C(A,B):
    # class C inheriting property of class A
    # class C inheriting property of class B
    # more properties of class C
```

Example

```
class Person:
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def getname(self):
        return self.name
    def getage(self):
        return self.age
class Student:
    def __init__(self,roll,per):
        self.roll=roll
        self.per=per
    def getroll(self):
        return self.roll
    def getper(self):
        return self.per
class ScienceStudent(Person,Student):

    def __init__(self,name,age,roll,per,stream):
        Person.__init__(self,name,age)
        Student.__init__(self,roll,per)
        self.stream=stream
    def getstream(self):
        return self.stream

ms=ScienceStudent("Suresh",19,203,89.4,"maths")
print("Name:",ms.getname())
print("Age:",ms.getage())
print("Roll:",ms.getroll())
print("Per:",ms.getper())
print("Stream:",ms.getstream())
```

Output:

```
Name: Suresh
Age: 19
Roll: 203
Per: 89.4
Stream: maths
```

Guess The Output ?

```
class A:
    def m(self):
        print("m of A called")

class B:
    def m(self):
        print("m of B called")

class C(A,B):
    pass

obj=C()
obj.m()
```

Output:

```
m of A called
```

Why did **m( )** of **A** got called ?

This is because of a special rule in **Python** called **MRO**

What Is MRO In Python ?

- In languages that use **multiple inheritance**, the order in which **base classes** are searched when looking for a **method** is often called the **Method Resolution Order**, or **MRO.**

- **MRO RULE :**
  - In the multiple inheritance scenario, any specified attribute is searched **first in the current class**. If not found, the search continues into **parent classes**, **left-right fashion** and **then in depth-first without searching same class twice.**

Can We See This MRO ?

- Yes, Python allows us to see this MRO by calling a method called **mro( )** which is present in every class by default.

Example

```python
class A:
    def m(self):
        print("m of A called")

class B:
    def m(self):
        print("m of B called")

class C(A,B):
    pass
print(C.mro())
```

Output

```
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'objec
t'>]
```

Another Way To See MRO ?

- There is a tuple also called __**mro**__ made available in **every class** by **Python** using which we can get the same output as before

Example

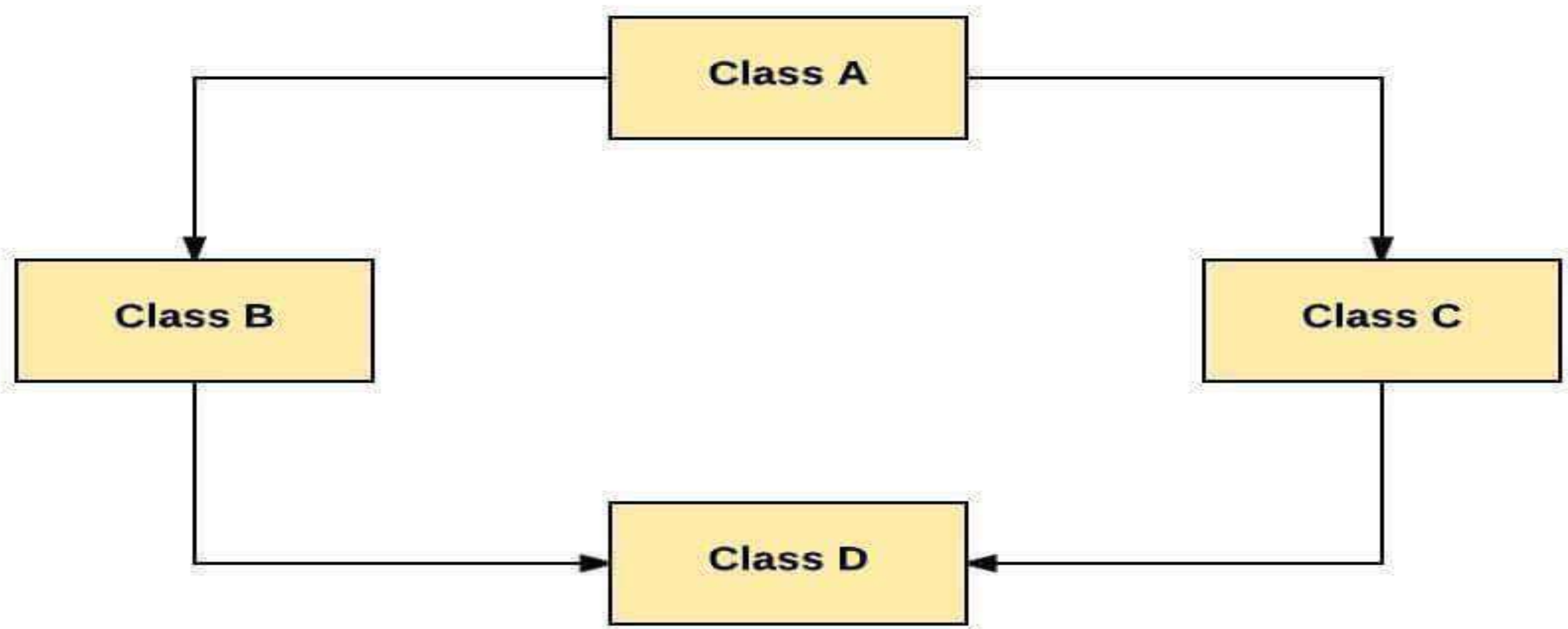
```
class A:
    def m(self):
        print("m of A called")

class B:
    def m(self):
        print("m of B called")

class C(A,B):
    pass
print(C.__mro__)
```

Output

```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'objec
t'>)
```

The Hybrid Inheritance

- This form combines more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.



Example

```python
class A:
    def m1(self):
        print("m1 of A called")
class B(A):
    def m2(self):
        print("m2 of B called")
class C(A):
    def m3(self):
        print("m3 of C called")
class D(B,C):
    pass
```

```python
obj=D()
obj.m1()
obj.m2()
obj.m3()
```

Output:

```
m1 of A called
m2 of B called
m3 of C called
```

The Diamond Problem

- The **"diamond problem"** is the generally used term for an **ambiguity** that arises in **hybrid inheritance** .

- Suppose two classes **B** and **C** inherit from a superclass **A**, and another class **D** inherits from both **B** and **C**.

- If there is a **method "m"** in **A** that **B** and **C** have overridden, then the question is **which version of the method does D inherit?**

Guess The Output

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    pass
```

```
obj=D()
obj.m()
```

Output:

```
m of B called
```

Why m() of B was called ?

As discussed previously , Python uses MRO to search for an attribute which goes from left to right and then in depth first. Now since B is the first inherited class of D so Python called m( ) of B

Guess The Output

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")

class C(A):
    def m(self):
        print("m of C called")

class D(C,B):
    pass
```

```
obj=D()
obj.m()
```

Output:

```
m of C called
```

Guess The Output

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    pass

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    pass
```

```
obj=D()
obj.m()
```

Output:

```
m of C called
```

Why m() of C was called ?

MRO goes from left to right first and then depth first. In our case Python will look for method m() in B but it won't find it there . Then it will search m() in C before going to A. Since it finds m() in C, it executes it dropping the further search

Guess The Output

```
class A:
   def m(self):
      print("m of A called")

class B(A):
 def m(self):
      print("m of B called")

class C(A):
   def m(self):
      print("m of C called")

class D(B,C):
 def m(self):
      print("m of D called")
```

```
obj=D()
obj.m()
```

<u>Output:</u>

```
m of D called
```

PYTHON
LECTURE 46

Today's Agenda

- **Exception Handling**
  - Introduction To Exception Handling
  - Exception Handling Keywords
  - Exception Handling Syntax
  - Handling Multiple Exceptions
  - Handling All Exceptions

What Is An Exception ?

- **Exception** are errors that occur at runtime .

- In other words , if our program encounters an **abnormal situation** during it's execution it **raises** an **exception**.

- **For example,** the statement

  a=10/0

will generate an **exception** because **Python** has no way to solve **division by 0**

What Python Does When An Exception Occurs ?

- Whenever an **exception** occurs , **Python** does 2 things :
  - **It immediately terminates the code**
  - **It displays the error message related to the exception in a technical way**
- Both the steps taken by **Python** cannot be considered user friendly because
  - **Even if a statement generates exception , still other parts of the program must get a chance to run**
  - **The error message must be simpler for the user to understand**

A Sample Code

```
a=int(input("Enter first no:"))
b=int(input("Enter second no:"))
c=a/b
print("Div is",c)
d=a+b
print("Sum is",d)
```

Output:

```
Enter first no:10
Enter second no:5
Div is 2.0
Sum is 15
```

```
Enter first no:10
Enter second no:0
Traceback (most recent call last):
  File "except1.py", line 3, in <module>
    c=a/b
ZeroDivisionError: division by zero
```

As we can observe , in the second run the code generated exception because Python does not know how to handle division by 0. Moreover it did not even calculated the sum of 10 and 0 which is possible

A Sample Code

```
a=int(input("Enter first no:"))
b=int(input("Enter second no:"))
c=a/b
print("Div is",c)
d=a+b
print("Sum is",d)
```

In this case since it is not possible for **Python** to covert **"2a"** into an **integer** , so it generated an **exception** . But the message it displays is **too technical** to understand

Output:

```
Enter first no:10
Enter second no:2a
Traceback (most recent call last):
  File "except1.py", line 2, in <module>
    b=int(input("Enter second no:"))
ValueError: invalid literal for int() with base 10: '2a'
```

How To Handle Such Situations ?

- If we want our program to behave **normally** , even if an **exception** occurs , then we will have to apply **Exception Handling**

- **Exception handling** is a mechanism which allows us to handle errors **gracefully** while the program is running instead of **abruptly ending** the program execution.

Exception Handling Keywords

- Python provides **5 keywords** to perform **Exception Handling:**
  - **try**
  - **except**
  - **else**
  - **raise**
  - **finally**

Exception Handling Syntax

- Following is the **syntax** of a **Python** **try-except-else** block.

try:

You do your operations here;

....

except ExceptionI:

If there is ExceptionI, then execute this block.

except ExceptionII:

If there is ExceptionII, then execute this block.

....

else:

If there is no exception then execute this block.

Remember !
In place of Exception I and Exception II , we have to use the names of Exception classes in Python

Improved Version Of Previous Code

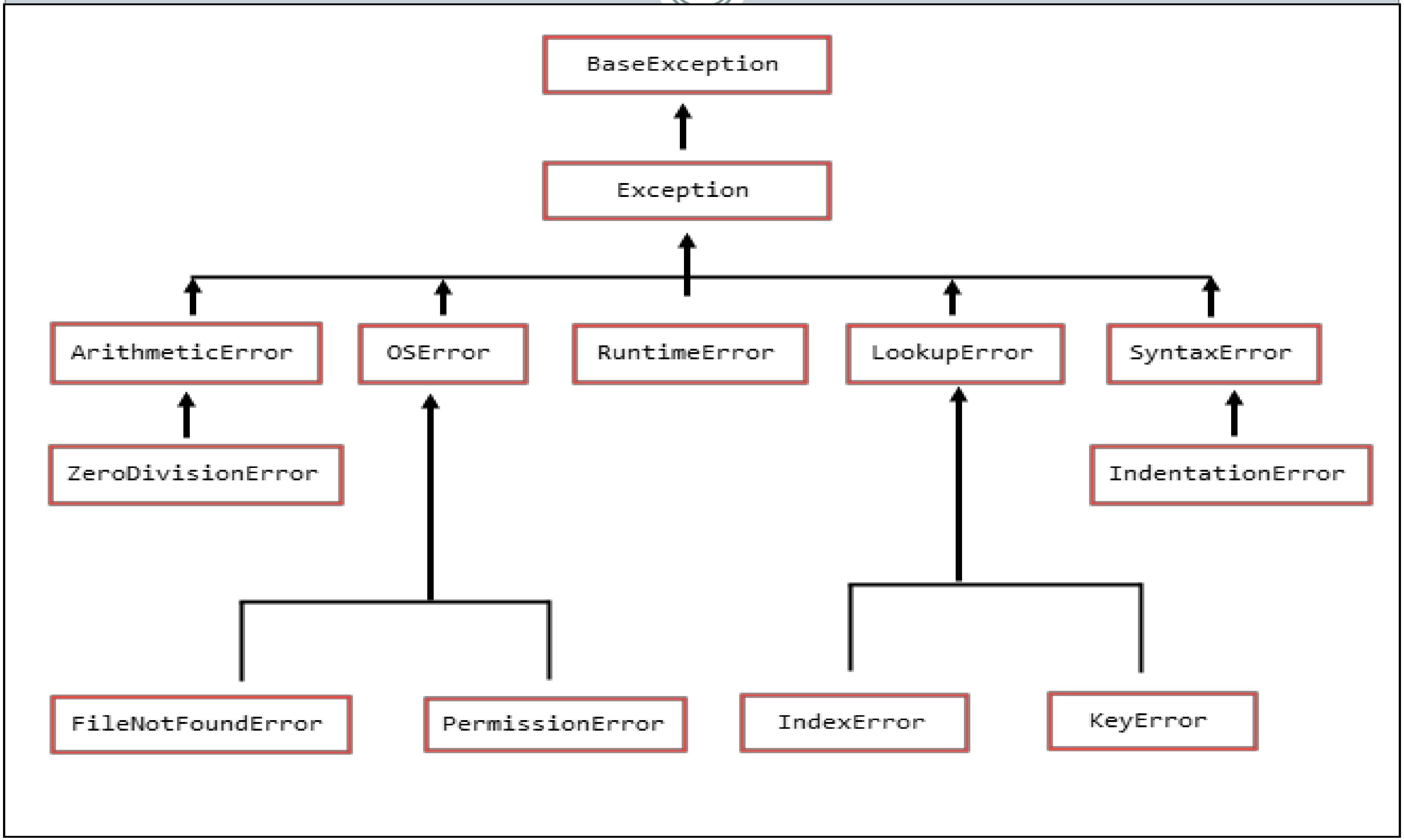
```
a=int(input("Enter first no:"))
b=int(input("Enter second no:"))
try:
    c=a/b
    print("Div is",c)
except ZeroDivisionError:
    print("Denominator should not be 0")
d=a+b
print("Sum is",d)
```

Output:

```
Enter first no:10
Enter second no:0
Denominator should not be 0
Sum is 10
```

```
Enter first no:10
Enter second no:3
Div is 3.3333333333333335
Sum is 13
```

Exception Hierarchy



Important Exception Classes

Exception Class	Description
Exception	Base class for all exceptions
ArithmeticError	Raised when **numeric calculations fails**
FloatingPointError	Raised when a **floating point calculation fails**
ZeroDivisionError	Raised when **division** or **modulo** by **zero** takes place for **all numeric types**
OverflowError	Raised when result of an **arithmetic operation is too large** to be represented
ImportError	Raised when the imported module is not found in **Python version < 3.6**
ModuleNotFoundError	Raised when the imported module is not found from **Python version >=3.6**

Important Exception Classes

Exception Class	Description
LookupError	Raised when **searching /lookup** fails
KeyError	Raised when the **specified key** is **not found** in the **dictionary**
IndexError	Raised when **index** of a **sequence is out of range**
NameError	Raised when an **identifier** is **not found** in the **local** or **global namespace**
UnboundLocalError	Raise when we use a **local variable** in a function **before declaring** it.
TypeError	Raised when a **function or operation** is applied to an **object of incorrect** type
ValueError	Raised when a function gets argument of correct type but improper value

Important Exception Classes

Exception Class	Description
AttributeError	Raised when a non-existent attribute is referenced.
OSError	Raised when system operation causes system related error.
FileNotFoundError	Raised when a file is not present
FileExistsError	Raised when we try to create a directory which is already present
PermissionError	Raised when trying to run an operation without the adequate access rights.
SyntaxError	Raised when there is an error in Python syntax.
IndentationError	Raised when indentation is not specified properly.

Handling Multiple Exception

- A **try** statement may have more than one **except** clause for **different exceptions.**

- But at most one **except** clause will be executed

Point To Remember

- Also , we must remember that if we are handling **parent and child exception classes** in **except** clause then the **parent exception** must appear **after child exception** , otherwise child except will never get a chance to run

Guess The Output !

```
import math
try:

    x=10/5
    print(x)
    ans=math.exp(3)
    print(ans)

except ZeroDivisionError:
    print("Division by 0 exception occurred!")
except ArithmeticError:
    print("Numeric calculation failed!")
```

Output:

```
2.0
20.085536923187668
```

Guess The Output !

```python
import math
try:

    x=10/0
    print(x)
    ans=math.exp(20000)
    print(ans)

except ZeroDivisionError:
    print("Division by 0 exception occurred!")
except ArithmeticError:
    print("Numeric calculation failed!")
```

<u>Output:</u>

```
Division by 0 exception occurred!
```

Guess The Output !

```
import math
try:

    x=10/5
    print(x)
    ans=math.exp(20000)
    print(ans)

except ZeroDivisionError:
    print("Division by 0 exception occurred!")
except ArithmeticError:
    print("Numeric calculation failed!")
```

Output:

```
2.0
Numeric calculation failed!
```

Guess The Output !

```
import math
try:

    x=10/5
    print(x)
    ans=math.exp(20000)
    print(ans)

except ArithmeticError:
    print("Numeric calculation failed!")
except ZeroDivisionError:
    print("Division by 0 exception occurred!")
```

Output:

```
2.0
Numeric calculation failed!
```

Guess The Output !

```
import math
try:

    x=10/0
    print(x)
    ans=math.exp(20000)
    print(ans)

except ArithmeticError:
    print("Numeric calculation failed!")
except ZeroDivisionError:
    print("Division by 0 exception occurred!")
```

Output:

```
Numeric calculation failed!
```

Exercise

- Write a program to ask the user to input 2 integers and calculate and print their division. Make sure your program behaves as follows:
  - If the user enters a non integer value then ask him to enter only integers
  - If denominator is 0 , then ask him to input non-zero denominator
  - Repeat the process until correct input is given
- Only if the inputs are correct then display their division and terminate the code

Sample Output

```
Input first no:10
Input second no:0
Please input non-zero denominator
Input first no:a
Please input integers only! Try again
Input first no:10
Input second no:a
Please input integers only! Try again
Input first no:4
Input second no:5
Div is  0.8
```

Solution

```
while(True):
    try:
        a=int(input("Input first no:"))
        b=int(input("Input second no:"))
        c=a/b
        print("Div is ",c)
        break
    except ValueError:
        print("Please input integers only! Try again")
    except ZeroDivisionError:
        print("Please input non-zero denominator")
```

Single **except**, Multiple Exception

- If we want to write a single **except** clause to handle **multiple exceptions** , we can do this .
- For this we have to write **names of all the exceptions** within **parenthesis** separated with **comma** after the keyword **except**

Example

```
while(True):
    try:
        a=int(input("Input first no:"))
        b=int(input("Input second no:"))
        c=a/b
        print("Div is ",c)
        break
    except (ValueError,ZeroDivisionError):
        print("Either input is incorrect or denominator is 0. Try again!")
```

Sample Output

```
Input first no:4
Input second no:0
Either input is incorrect or denominator is 0. Try again!
Input first no:10
Input second no:bhopal
Either input is incorrect or denominator is 0. Try again!
Input first no:10
Input second no:4
Div is  2.5
```

Handling All Exceptions

- We can write the keyword **except** without any **exception class name** also .

- In this case for every **exception** this except clause will run .

- The only problem will be that we will never know the **type of exception** that has occurred!

Exception Handling Syntax

- Following is the <u>**syntax**</u> of a **Python handle all exception** block.

try:

You do your operations he

....

except :

For every kind of exception this block will execute

Notice , we have not provided any name for the exception

Example

```
while(True):
try:
a=int(input("Input first no:"))
b=int(input("Input second no:")) c=a/b
print("Div is ",c) break
except:
print("Some problem occurred. Try again!")
```

Sample Output

```
Input first no:10
Input second no:0
Some problem occurred. Try again!
Input first no:10
Input second no:a
Some problem occurred. Try again!
Input first no:10
Input second no:4
Div is  2.5
```

PYTHON

LECTURE 47

Today’s Agenda

- **Exception Handling**
  - Using Exception Object
  - Getting Details Of Exception
  - Raising An Exception
  - Using finally Block
  - Creating User Defined Exceptions

Using Exception Object

- Now we know how to handle exception, in this section we will learn how to access **exception object** in exception handler code.

- To access the **exception object** created by Python we can use the keyword **as** and assign it to a **variable**.

- Finally using that variable we can get the details of the exception

Example

```
while(True):
    try:
        a=int(input("Input first no:"))
        b=int(input("Input second no:"))
        c=a/b
        print("Div is ",c)
        break;
    except (ValueError,ZeroDivisionError) as e:
        print(e)
```

Sample Output

```
Input first no:10
Input second no:0
division by zero
Input first no:10
Input second no:a
invalid literal for int() with base 10: 'a'
Input first no:10
Input second no:5
Div is  2.0
```

Obtaining Exception Details Using traceback class

- Sometimes , we need to print the details of the exception exactly ***like Python does*** .

- We do this normally , when we are **debugging our code**.

- The module **traceback** helps us do this

Obtaining Exception Details Using traceback module

- This module contains a function called **format_exc( )**

- It returns **complete details** of the exception as a **string.**

- This **string** contains:
  - The **program name** in which **exception** occurred
  - **Line number** where **exception** occurred
  - The **code** which generated the **exception**
  - The **name** of the **exception class**
  - The **message** related to the **exception**

Example

```
import traceback
while(True):
    try:
        a=int(input("Input first no:"))
        b=int(input("Input second no:"))
        c=a/b
        print("Div is ",c)
        break
    except:
        print(traceback.format_exc())
```

Sample Output

```
Input first no:10
Input second no:0
Traceback (most recent call last):
  File "except5.py", line 6, in <module>
    c=a/b
ZeroDivisionError: division by zero

Input first no:10
Input second no:bhopal
Traceback (most recent call last):
  File "except5.py", line 5, in <module>
    b=int(input("Input second no:"))
ValueError: invalid literal for int() with base 10: 'bhopal'

Input first no:10
Input second no:5
Div is  2.0
```

Raising An Exception

- We can force **Python** to generate an **Exception** using the keyword **raise**.

- This is normally done in those situations where we want **Python** to throw an exception in a particular condition of our choice

- **Syntax:**
  - **raise ExceptionClassName**
  - **raise ExceptionClassName(message)**

Exercise

- Write a program to ask the user to input 2 integers and calculate and print their division. Make sure your program behaves as follows:

  - If the user enters a non integer value then ask him to enter only integers
  - If denominator is 0 , then ask him to input non-zero denominator
  - **If any of the numbers is negative or numerator is 0 then display the message negative numbers not allowed**
  - Repeat the process until correct input is given

- Only if the inputs are correct then display their division and terminate the code

Sample Output

```
Input first no:10
Input second no:-4
Negative numbers not allowed!Try again
Input first no:10
Input second no:0
Please input non-zero denominator
Input first no:-1
Input second no:4
Negative numbers not allowed!Try again
Input first no:10
Input second no:bhopal
Please input integers only! Try again
Input first no:20
Input second no:5
Div is  4.0
```

Solution

```
while(True):
    try:
        a=int(input("Input first no:"))
        b=int(input("Input second no:"))
        if a<=0 or b<0:
                raise Exception("Negative numbers not allowed!Try again")
        c=a/b
        print("Div is ",c)
        break;
    except ValueError:
        print("Please input integers only! Try again")
    except ZeroDivisionError:
        print("Please input non-zero denominator")
    except Exception as e:
        print(e)
```

The **finally** Block

- If we have a code which we want to run in all situations, then we should write it inside the **finally** block.
- **Python** will always run the instructions coded in the **finally** block.
- It is the most common way of doing **clean up tasks** , like, **closing a file** or **disconnecting with the DB** or **logging out the user** etc

Syntax Of The finally Block

- The **finally** block has 2 syntaxes:

Syntax 1

```
try:
    # some exception generating code
except :
    # exception handling code
finally:
    # code to be always executed
```

Syntax 2

```
try:
        # some exception generating code
finally:
        # code to be always executed
```

Guess The Output ?

```
while(True):
    try:
        a=int(input("Input first no:"))
        b=int(input("Input second no:"))
        c=a/b
        print("Div is ",c)
        break;
    except ZeroDivisionError:
        print("Denominator should not be zero")
    finally:
        print("Thank you for using the app!")
```

Output:

```
Input first no:10
Input second no:0
Denominator should not be zero
Thank you for using the app!
Input first no:10
Input second no:5
Div is  2.0
Thank you for using the app!
```

```
Input first no:10
Input second no:a
Thank you for using the app!
Traceback (most recent call last):
  File "except8.py", line 4, in <module>
    b=int(input("Input second no:"))
ValueError: invalid literal for int() with base 10: 'a'
```

Creating User Defined Exception

- Python has many **built-in exceptions** which forces our program to output an error when something in it goes wrong.
- However, sometimes we may need to create our own exceptions which will be more suitable for our purpose.
- Such exceptions are called **User Defined Exceptions**

Creating User Defined Exception

- In **Python**, users can define such exceptions by creating a **new class**.
- This **exception class** has to be **derived**, either directly or indirectly, from **Exception** class.
- Most of the **built-in exceptions** are also derived form this class.