# Unit-5
# Object oriented programming with Python

Prof. Manan Thakkar
Assistant Professor, Dept. of Computer Engg.
UVPCE, Ganpat University, Mehsana

# OOPs concepts

- Python is an object-oriented programming language.
- Major principles of object-oriented programming system are given below.
- Object
- Class
- Encapsulation
- Inheritance
- Polymorphism
- Data Abstraction
- Data Hiding

# What is OOP?

- Object-oriented Programming, or OOP is a programming paradigm which provides a means of structuring programs so that attributes (variable) and behaviors (methods) are bundled into individual objects.
- Fundamental entities of OOP are: class & object
- object-oriented programming is an approach for modeling concrete, real-world things like cars as well as relations between things like companies and employees, students and teachers, etc.
- OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.

# Encapsulation & Data hiding

- **Encapsulation** means wrapping the implementation of data member (variables) and methods inside a class. Encapsulation concerns about wrapping data to hide the complexity of a system.
- Encapsulated method tells **what** action it performs on object, but it does not describe **how** does it perform that action.
- In OOP, encapsulation is achieved **through class and object.**
- **Data Hiding** means restricting the use of members of a class to prevent an illegal or unauthorized access.
- The main difference between data hiding and encapsulation is that data hiding focus more on **data security** and encapsulation focuses more on **hiding the complexity of the system**.

# Object

- **Object:** It is an entity that has attributes and behaviour.
- For example, Ram is an object who has attributes (variables) such as height, weight, color etc. and has certain behaviors (methods) such as walking, talking, eating etc.
- **Everything in Python is an object**, and almost everything has attributes and methods. All functions have a built-in attribute __doc__, which returns the doc string defined in the function source code.

# Class

- **Class:** It is a blueprint of the objects.
- Class is collection of objects.
- For example, Ram, Steve, Aman are all objects so we can define a template (blueprint) class Human for these objects. The class can define the common attributes and behaviors of all the objects.
- Syntax:

```
class ClassName:
        <statement-1>
        .
        <statement-N>
```

- **Note:** Object is also called as instance of class.

# Class & object

- Syntax for creating an object:

Obj_name = classname(parameters-list)

- Example of program using class & object:

```
class parrot:
    species = "bird“       # class attribute
blu = parrot()          # instantiate the parrot class
```

- If you want to create an empty class, the syntax will be:

```
Class human:
    pass
```

# __init__() method (Constructor of a class)

- Methods that begins with double underscore __ are called special methods.
- The __init__ method is similar to constructors in C++ and Java.
- It is a special method in python which provides features of a constructor.
- We can pass any number of arguments while creating object, and it must match with __**init**__ definition.
- This method is mainly used to initialize the objects.

- **Syntax:** Let's create parameter-less constructor

```
class Student:
    def __init__(self):
        print("This is non parametrized constructor")
s1 = Student()
```

# Cntd…

- Following example shows __init__() method is used to initialize object. Here, __init__() is also called as parameterized constructor:

```
class Employee:
    def __init__(self, name, id):
        self.id = id
        self.name = name
emp1 = Employee("John",101)
```

# __new__() v/s __init__()

- Whenever a class is instantiated __new__ and __init__ methods are called.
- **Special method __new__** will be called when **an object is created** and **__init__** method will be called to **initialize the object**.
- Use __new__ when you need to control the creation of a new instance.
- Use __init__ when you need to control initialization of a new instance.
- __new__ is the first step of instance creation. It's called first, and is responsible for returning a new instance of your class.
- In contrast, __init__ doesn't return anything; it's only responsible for initializing the instance after it's been created.
- In general, you shouldn't need to override __new__ unless you're subclassing an immutable type like str, int, unicode or tuple.

# Method

- Methods are functions written inside class. Methods are called by an object of class.
- One parameter is mandatory for method named- self. Consider following example:

```
class Employee:
    def __init__(self, name, id):
        self.id = id
        self.name = name
    def display (self):
        print("ID: %d \nName: %s"%(self.id, self.name))
    def show(self):
        return (self.id,self.name)
emp1 = Employee("John",101)
emp2 = Employee("David",102)
emp1.display()
a,b=emp2.show()
print(a,b)
```

Output of code:

```
ID: 101
Name: John
102 David
```

# Cntd…

- Rewriting previous program without __init__() method:

```
class Employee:
    def display (self,id,name):
        self.ids=id
        self.names=name
        print("ID:{0}  Name:{1}".format(self.ids, self.names))
    def show(self):
        return (self.ids, self.names)
emp1 = Employee()
emp1.display(5,'john')
a,b=emp1.show()
print(a,b)
```

**Output:**

```
ID:5  Name:john
5 john
```

# Instance variable & Static variable

- Instance variable is a variable which can be accessed through object of class.
- Instance is assigned inside a constructor or method with self.
- Static variable is a variable which can be accessed through class name.
- Static variable can be assigned anywhere in class. It is initialized only once when the class is loaded.
- **Example:** In following example *value* is instance variable & *count* is class variable

```
class Student:
    count = 0
    def __init__(self, val):
        self.value=val
        print("instance variable:",self.value)
        Student.count = Student.count + 1
s1=Student('a')
s2=Student('b')
print("The number of students:",Student.count)
```

Output:
instance variable: a
instance variable: b
The number of students: 2

# Modifying object properties

- Object properties can be accessed/modified in main program.
- Consider following example.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
p1.age = 40
print(p1.age)
```

**Output:** 40

# Deleting object properties

- Consider following example:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
del p1.age
print(p1.age)
```

**Output:** AttributeError: 'Person' object has no attribute 'age'

# Deleting an object

- Deletion of an object can be done using ‘del’ keyword. For example,

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age
p1 = Person("John", 36)
del p1
print(p1)
```

**Output:** NameError: name 'p1' is not defined

# Destructor

- A **destructor** is used to destroy the object and perform the final clean up.
- In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically.
- The **special method __del__()** is a known as a destructor method in Python.
- It is called when ‘del’ keyword is used in main program.
- Thus, object is created using __new__() method and destroyed using __del__() method.
- Syntax of destructor declaration :

```
def __del__(self):
  # body of destructor
```

# Cntd…

- Example-1:

```
class Employee:
    def __init__(self):
        print('object initialized.')
    def __del__(self):
        print('Destructor called, object deleted.')
obj = Employee()
del obj
```

**Output:**

```
object initialized.
Destructor called, object deleted.
```

# Static method / class method

- Just like static variables, static methods are the methods which are bound to the class rather than an object of the class and hence are called using the class name.
- As static methods are bound to the class hence they cannot change the state of an object.
- To call a static method we don't need any class object it can be directly called using the class name.
- In python there are two ways of defining a static method:

1) Using the staticmethod()
2) Using the @staticmethod

# Cntd…

1) Using staticmethod():

Consider following example:

```
class Shape:
    def info(msg):
        print(msg)
        print("Representing different shapes.")
Shape.info = staticmethod(Shape.info)
Shape.info("Welcome to Shape class")
```

**Output:**

```
Welcome to Shape class
Representing different shapes.
```

# Cntd…

1) Using @staticmethod:

Here, @ is called as python **decorator**.

Consider following example:

```
class Shape:
    @staticmethod
    def info(msg):
        print(msg)
        print("Representing different shapes.")
Shape.info("Using @staticmethod")
```

**Output:**

```
Using @staticmethod
Representing different shapes.
```

# Important points

- Instance variables and methods are accessed by objects of a class in which these variable and methods are defined.
- Static variable and methods can be only accessed by class name. They are used when we want to define some behaviour or property specific to the class and which is something common for all the class objects.
- If you look closely, for a static method we don't provide the argument **self** because static methods don't operate on objects.

# Built-in class functions

- Built-in functions which are used for class are listed below:

| SrNo | Function | Description |
|---|---|---|
| 1 | getattr(obj,name,default) | It is used to access the attribute of the object. |
| 2 | setattr(obj, name,value) | It is used to set a particular value to the specific attribute of an object. |
| 3 | delattr(obj, name) | It is used to delete a specific attribute. |
| 4 | hasattr(obj, name) | It returns true if the object contains some specific attribute. |

- Let us take an example to understand all of these functions.

```
class Student:
    def __init__(self,name,id,age):
        self.name = name
        self.id = id
        self.age = age
#creates the object of the class Student
s = Student("John",101,22)
#prints the attribute name of the object s
print(getattr(s,'name'))
# reset the value of attribute age to 23
setattr(s,"age",23)
# prints the modified value of age
print(getattr(s,'age'))
# prints true if the student contains the attribute with name id
print(hasattr(s,'id'))
# deletes the attribute age
delattr(s,'age')
# this will give an error since the attribute age has been deleted
print(s.age)
```

**Output:**

```
John
23
True
AttributeError: 'Student' object has no attribute 'age'
```

# Built-in class attributes

- A class contains some built-in class attributes which provide information about the class.
- The built-in class attributes are given in the below table.

| SrNo | Attribute | Description |
|---|---|---|
| 1 | __dict__ | It provides the dictionary containing the information about the class namespace. |
| 2 | __doc__ | It contains a string which has the class documentation |
| 3 | __name__ | It is used to access the class name. |
| 4 | __module__ | It is used to access the module in which, this class is defined. |
| 5 | __bases__ | It contains a tuple including all base classes. |

Consider following example to understand class attributes:

```
class Student:
    def __init__(self,name,id):
        self.names = name
        self.ids=id
    def display(self):
        print("Python class attributes")
s = Student("John",101)
print(Student.__doc__)
print(Student.__dict__)
print(s.__dict__)
print(Student.__module__)
print(Student.__bases__)
```

# Cntd…

- Output of previous program:

None

{'__module__': '__main__', '__init__': <function Student.__init__ at 0x0000019C6AFE5E58>, 'display': <function Student.display at 0x0000019C6AFE5F78>, '__dict__': <attribute '__dict__' of 'Student' objects>, '__weakref__': <attribute '__weakref__' of 'Student' objects>, '__doc__': None}

{'names': 'John', 'ids': 101}

__main__

(<class 'object'>,)

# Python program structure

- Consider an example:

1) s = Student("John",101,22)

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

**Output:** NameError: name 'Student' is not defined

2) temp=display()

```
def display():
    c="hello"
    return c
```

**Output:** NameError: name 'display' is not defined

# Cntd…

- Python execution steps:

1) Python first sets value of __name__ equal to '__main__'.
2) Then it runs 'def' & 'class' statements.
3) After that, it starts program execution.

- In previous program-1, first line tries to create an object 's' of class Student. But till this point python interpreter is unknown about Student class, hence it gives an error. Similar thing happened in previous program-2, where first line tries to call a function display(), but till this point python interpreter is unknown about definition of that function.

# Access Modifiers (Access specifiers)

- As we know that data hiding focus more on data security, to achieve that access modifiers are very useful.
- There are 3 types of access specifiers (access modifiers):

1) Public
2) Private
3) Protected

- Python **doesn't have exact** mechanism that **effectively restricts** access to any instance variable or method.
- Python has a convention of prefixing the name of the variable/method with single or double underscore to show the protected and private access specifiers. Public members are shown without any underscore.

# Public

- **Public :** The members declared as Public are accessible from outside the Class through an object of the class.
- All members in a Python class are **public** by default. Any member can be accessed from outside the class environment. Public variables are written without **any underscore** .
- Example:

```
class employee:
    def __init__(self, name, sal):
        self.name=name
        self.salary=sal
e1=employee("Kiran",10000)
print(e1.salary)
e1.salary= 20000
print(e1.salary)
```

**Output:**

```
10000
20000
```

# Protected

**Protected:** The members declared as Protected are only accessible in a class which is derived from it.

- That means protected members can be used in a class and its subclass (child class).
- To make an instance variable **protected,** add a single underscore (_) as a prefix to it.
- But python doesn’t have strict implementation of ‘protected’ access specifier, means it doesn't prevent instance variables being accessed outside of parent class or child class.
- It is programmer’s responsibility **not** to use such protected instance variables other than base and child class.

# Cntd…

- Example:

```
class employee:
    def __init__(self, name, sal):
        self._name=name   # protected attribute
        self._salary=sal      # protected attribute
e1=employee("Swati", 10000)
print(e1._name)
e1._name="Jinal“          #This is possible, but must not be used
print(e1._name)
```

**Output:**

```
Swati
Jinal
```

# Private

- Private members are only accessible within the class. No outside access is allowed.
- A variable with prefix of double underscore (__) makes it **private**. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError.
- Still there is a tricky way to access private members outside of its class. **How?**
- Python performs name mangling of private variables. So, every member with double underscore will be changed in format: object._class__variable
- Using above format, private variable can be accessed outside of its class. But it must be strictly avoided, unless inevitable.

# Cntd…

- Example:

```
class employee:
    def __init__(self, name, sal):
        self.__name=name            # private attribute
        self.__salary=sal          # private attribute
e1=employee("Bill",10000)
print(e1._employee__salary)    #It must be avoided
e1._employee__salary=12000  #It must be avoided
print(e1._employee__salary)
print(e1.__salary)
```

**Output:**

```
10000
12000
AttributeError: 'employee' object has no attribute '__salary'
```

# Method Overloading

- It is called as **compile time** polymorphism. **Unlike** other languages ( C++, Java, C# etc.) , python **does not support** method overloading. We may overload the methods but can only use the latest defined method. For Example,

```
class test:
    def product(self,a, b):
        self.p=a*b
    def product(self,a, b, c):
        self.p=a*b*c
t1= test()
t1.product(4,5)    #This line produces Error
t1.product(4,5,3)
print(t1.p)
```

**Output:** TypeError: product() missing 1 required positional argument: 'c'

# Cntd…

- Reason for error in previous program is, product method with 2 parameters was replaced with product method with 3 parameters. Hence, ‘t1.product(4,5)’ statement gives an error because there is **no such method** named product having 2 parameter exist.
- If you comment line- ‘t1.product(4,5)’ then, statement ‘t1.product(4,5,3)’ will execute without any error because the latest version of product method contains 3 parameters. Because of this, method overloading is not possible in python.
- But there are some ways to achieve functionality like method overloading:
- 1) Using default values
- 2) Using variable length arguments
- **Note:** These ways gives you functionality like method overloading, but it is not actual method overloading. In method overloading, which method to call is determined at compile time, so, it is called as **static binding** or **early binding.**

**1) Using default values:** Consider following example

```
class Human:
    def greeting(self, fname=None,lname=None):
        if fname and lname is not None:
            print('Hello ' + fname+' '+lname)
        elif fname is not None:
            print('Hello '+ fname)
        else:
            print('Hello ')
obj = Human()
obj.greeting()
obj.greeting('Guido')
obj.greeting('Guido','Rossum')
```

**Output:**

```
Hello
Hello Guido
Hello Guido Rossum
```

**2) Using variable length arguments:** Consider following example

```
def add(instanceOf,*args):
    if instanceOf=='int':
        result=0
    if instanceOf=='str':
        result=''
    for i in args:
        result+=i
    return result
print(add('int',3,4,5,6,2))
print(add('str','I ','speak ','Gujarati'))
```

**Output:**

20

I speak Gujarati

# Operator Overloading

- Python operators work for built-in datatypes. But same operator behaves differently with different datatypes. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings.
- This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.
- In python, Special functions are used to perform operator overloading.
- When we use an operator then automatically a special function or magic function associated with that operator is invoked.
- Changing the behaviour of operator is as simple as changing the behaviour of method or function. You define methods in your class and operators work according to that behaviour defined in methods.
- When we use + operator, the **special method __add__** is automatically invoked in which the operation for + operator is defined. Now by changing this magic method's code, we can give extra meaning to the + operator.

# Overloading of binary + operator

- To overload binary + operator, we will need to implement __add__() function in the class. For Example,

```
class A:
    def __init__(self, a):
        self.a = a
    def __add__(self, o):
        return self.a + o.a
ob1 = A(1)
ob2 = A(2)
ob3 = A("python ")
ob4 = A("program")
u=ob1+ob2
v=ob3+ob4
print(u)
print(v)
```

**Output:**

```
3
python program
```

# Cntd…

- Example-2

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def __add__(self,other):
        x = self.x + other.x
        y = self.y + other.y
        return (x,y)
p1 = Point(2,3)
p2 = Point(-1,2)
print(p1 + p2)
```

**Output:** (1, 5)

• Example-3: Let us re-write example-2, using **special** function __str__()

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def __str__(self):
        return "({0},{1})".format(self.x, self.y)
    def __add__(self,other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x,y)
p1 = Point(2,3)
p2 = Point(-1,2)
print(p1 + p2)
```

**Output:** (1,5)

# Cntd…

- What actually happens is that, when you do p1 + p2, Python will call p1.__add__(p2) which in turn becomes Point.__add__(p1,p2).
- Like, '+' operator, we can overload other arithmetic operators like -, *, /, **, //, % etc. Following are the special (magic) methods used for arithmetic operators:

| Operator | Special (magic) Method |
|---|---|
| + | __add__(self, other) |
| – | __sub__(self, other) |
| * | __mul__(self, other) |
| / | __truediv__(self, other) |
| // | __floordiv__(self, other) |
| % | __mod__(self, other) |
| ** | __pow__(self, other) |

# Overloading of > operator

- Like arithmetic operators, we can overload relational (comparison) operators as well. For example,

```
class A:
    def __init__(self, a):
        self.a = a
    def __gt__(self, other):
        if(self.a>other.a):
            return True
        else:
            return False
ob1 = A(2)
ob2 = A(3)
if(ob1>ob2):
    print("ob1 is greater than ob2")
else:
    print("ob2 is greater than ob1")
```

**Output:** ob2 is greater than ob1

# Special (magic) methods for overloading of comparison & Assignment operators

| Comparison Operators | |
|---|---|
| Operator | Special Method |
| < | __lt__(self, other) |
| > | __gt__(self, other) |
| <= | __le__(self, other) |
| >= | __ge__(self, other) |
| == | __eq__(self, other) |
| != | __ne__(self, other) |

| Assignment Operators | |
|---|---|
| Operator | Magic Method |
| -= | __isub__(self, other) |
| += | __iadd__(self, other) |
| *= | __imul__(self, other) |
| /= | __idiv__(self, other) |
| //= | __ifloordiv__(self, other) |
| %= | __imod__(self, other) |
| **= | __ipow__(self, other) |

# Special (magic) methods for overloading of Logical & Unary operators

| Logical Operators | |
|---|---|
| Operator | Special method |
| << | __lshift__(self,other) |
| >> | __rshift__(self,other) |
| & | __and__(self,other) |
| \| | __or__(self,other) |
| ^ | __xor__(self,other) |
| ~ | __invert__(self) |

| Unary Operators | |
|---|---|
| Operator | Special Method |
| – | __neg__(self) |
| + | __pos__(self) |
| ~ | __invert__(self) |

# Inheritance

- Inheritance is the capability of one class to derive or inherit the properties from some another class. The benefits of inheritance are:

1) It represents real-world relationships well.
2) It provides **reusability** of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3) It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

- **Parent class** is the class being inherited from, also called base class or super class.
- **Child class** is the class that inherits from another class, also called derived class or sub class.
- **NOTE:** In python 3, Object is root class of all classes.

# Cntd…

- Example-1:

```
class Animal:           #Parent class
    def speak(self):
        print("Animal Speaking")
class Dog(Animal):  #Child class
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

**Output:**

```
dog barking
Animal Speaking
```

• Example-2:

```
class Person:
    def __init__(self, fname, lname):
        self.fn = fname
        self.ln = lname
class Student(Person):
    def __init__(self, fname, lname,city):
        Person.__init__(self, fname, lname)
        self.mycity=city
    def printname(self):
        print(self.fn, self.ln,self.mycity)
x = Student("Mike", "Olsen","Paris")
x.printname()
```

**Output:** Mike Olsen Paris

Example-3 : Re-writing example-2 with super() function

```
class Person:
    def __init__(self, fname, lname):
        self.fn = fname
        self.ln = lname
class Student(Person):
    def __init__(self, fname, lname,city):
        super().__init__(fname, lname)
        self.mycity=city
    def printname(self):
        print(self.fn, self.ln,self.mycity)
x = Student("Mike", "Olsen","Paris")
x.printname()
```

**Output:** Mike Olsen Paris

• Example-4:

```
class Person:
  def __init__(self, fname, lname):
    self.fn = fname
    self.ln = lname
  def printname(self):
    print(self.fn, self.ln)

class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.passingyear = year
  def welcome(self):
    print("Welcome", self.fn, self.ln, "to class of", self.passingyear)
x = Student("Mike", "Olsen", 2019)
x.printname()
x.welcome()
```

**Output:** Mike Olsen
Welcome Mike Olsen to class of 2019

# Types of inheritance

1) Single Inheritance
2) Multilevel Inheritance
3) Multiple Inheritance
4) Hierarchical Inheritance
5) Hybrid Inheritance

**1) Single Inheritance:** When a child class inherits only a single parent class. Example,

```
class Parent:
    def func1(self):
        print("this is function one")
class Child(Parent):
    def func2(self):
        print(" this is function 2 ")
ob = Child()
ob.func1()
ob.func2()
```

# Cntd…

**2) Multilevel Inheritance:** When a child class becomes a parent class for another child class. Example,

```
class Animal:
    def speak(self):
        print("Animal Speaking")
class Dog(Animal):
    def bark(self):
        print("dog barking")
class BabyDog(Dog):
    def eat(self):
        print("Eating bread...")
d = BabyDog()
d.bark()
d.speak()
d.eat()
```

**Output:** dog barking
Animal Speaking
Eating bread...

# Cntd…

**3) Multiple Inheritance:** When a child class inherits from more than one parent class.

```
class Father:
    def show_father(self,dad):
        self.f_name=dad
class Mother:
    def show_mother(self,mom):
        self.m_name=mom
class Son(Father, Mother):
    def show_parent(self):
        print("Father:",self.f_name,"Mother:",self.m_name)
s1 = Son()  # Object of Son class
s1.show_father("Mark")
s1.show_mother("Sonia")
s1.show_parent()
```

**Output:** Father: Mark Mother: Sonia

# Cntd…

**4) Hierarchical Inheritance:** One class is inherited from second class and second class is inherited from third class and so on.

```
class Father:
    def show_father(self,dad):
        self.f_name=dad
        return self.f_name
class Son(Father):
    def show_son(self):
        pass
class Daughter(Father):
    def show_daughter(self):
        pass
s1 = Son(); d1=Daughter()  # Objects of Father class
print(s1.show_father("Rasik"))
print(d1.show_father('Prakash'))
```

**Output:**

Rasik

Prakash

**5) Hybrid Inheritance:** Hybrid inheritance is a combination of multiple and multilevel inheritance. More than one child class inherit a parent class. Those child classes, in turn, acts as the parent class for another class. For Example,

```
class Parent:
    def func1(self):
        print("parent class")
class Child1(Parent):
    def func2(self):
        print("child class 1")
class Child2(Parent):
    def func3(self):
        print("child class 2")
class Grandchild(Child1 , Child2):
    def func4(self):
        print("grand child")
ob =Grandchild()
ob.func1()
```

**Output:**

parent class

# Method Overriding

- When the same parent class method is defined in the child class with some specific implementation, then the concept is called method overriding.
- We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.
- Base class method is said to be **‘overridden’** by child class method.
- It shows **Run-time polymorphism.**
- Here, which method to call is determined at runtime, this is called **dynamic binding** or **late binding.**

# Cntd…

- Example-1:

```
class Animal:
    def speak(self):
        print("In parent class")
class Dog(Animal):
    def speak(self):
        print("In child class")
d1= Dog()
d1.speak()
```

**Output:**

```
In child class
```

Example-2:

```
class Bank:
    def getroi(self):
        return 10
class SBI(Bank):
    def getroi(self):
        return 7
class ICICI(Bank):
    def getroi(self):
        return 8
b1= SBI()
b2= ICICI()
print("SBI Rate of interest:",b1.getroi())
print("ICICI Rate of interest:",b2.getroi())
```

**Output:**

SBI Rate of interest: 7
ICICI Rate of interest: 8

# Method Resolution Order

- Method Resolution Order(MRO) it denotes the way a programming language resolves a method or attribute.
- In python, method resolution order defines- which order interpreter must follow to search base classes, when executing a method.
- While inheriting from another class, the interpreter needs a way to resolve the methods with same name that are being called via an instance. Thus we need the method resolution order.
- This order is also called Linearization of a class and set of rules used to find this order. First, the method or attribute is searched within a class and then it follows the order we specified while inheriting from left to right and depth first order.
- **Thumb rule:** A parent class can not come in linear sequence (linearization) until its all child classes have been visited to search a method.
- **Note:** by default, all class methods are virtual in python

Example-1:

```
class A:
    def rk(self):
        print("class A")
class B(A):
    def rk(self):
        print("class B")
class C(A):
    def rk(self):
        print("class C")
class D(B,C):
    pass
r=D()
r.rk()
```

**Output:**

```
class B
```

Example-2:

```
class A:
    def rk(self):
        print("class A")
class T:
    def rk(self):
        print("class T")
class B(T):
    pass
class C(A):
    def rk(self):
        print("class C")
class D(B,C):
    pass
r=D()
r.rk()
```

**Output:** class T

# Built-in functions used in Inheritance

1) issubclass (sub, sup) method:

It is used to check the relationships between the specified classes. It returns true if the first class is the subclass of the second class, and false otherwise.

• Example:

```
class Calculation1:
    def Summation(self,a,b):
        return a+b
class Calculation2:
    def Multiplication(self,a,b):
        return a*b
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b
d=Derived()
print(issubclass(Derived,Calculation2))
print(issubclass(Calculation1,Calculation2))
```

**Output:**

True

False

# Cntd…

2) isinstance (obj, class) method

It is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

• Example:

```
class Calculation1:
    def Summation(self,a,b):
        return a+b
class Calculation2:
    def Multiplication(self,a,b):
        return a*b
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b
d=Derived()
print(isinstance(d,Derived))
print(isinstance(d,Calculation1))
```

**Output:**

```
True
True
```

# Exception handling

- An exception can be defined as an abnormal condition in a program which causes the program to terminate.
- **Exception** is the base class for all the exceptions in python.
- ZeroDivisionError, NameError, IndentationError, IOError, ValueError, TypeError and IndexError are very common exceptions.
- Using exception handling mechanism we can stop abnormal termination of program, hence, program terminates in a normal mode.

# Cntd…

- A list of common exceptions that can be thrown from a normal python program is given below.

1) ZeroDivisionError: Occurs when a number is divided by zero.
2) NameError: It occurs when a name is not found. It may be local or global.
3) IndentationError: If incorrect indentation is given.
4) IOError: It occurs when Input Output operation fails.
5) ValueError: It occurs when invalid value is given as a parameter for a function
6) TypeError: It occurs when some operation is performed between operand of unsupported data types.
7) IndexError: It is raised when index of a sequence is not found.

# Structure of exception handling block-1

try:

Write code here which may generate an exception

except:

If there is any exception, then this block will be automatically executed.

else:

If there is no exception then execute this block.

# Structure of exception handling block-2

try:

Write code here which may generate an exception

except Exception-1:

If there is Exception-1, then this block will be automatically executed.

except Exception-2:

If there is Exception-2, then this block will be automatically executed.

.

.

.

except Exception-N:

If there is Exception-N, then this block will be automatically executed.

else:

If there is no exception then execute this block.

# Structure of exception handling block-3

try:

Write code here which may generate an exception

except (Exception-1, Exception-2, …,Exception-N)

If there is any exception from the given exception list, then execute this block.

else:

If there is no exception then execute this block.

# Structure of exception handling block-4

try:

Write code here which may generate an exception

except:

If there is any exception, then this block will be automatically executed.

else:

If there is no exception then execute this block.

finally:

This would **always** be executed, even if **unhandled exception** occur.

# Structure of exception handling block-5

try:

Write code here which may generate an exception

finally:

This would **always** be executed, even if **unhandled exception** occur.

# Examples

```
try:
    c = 6/0;
    print("c = %d"%c)
except:
    print("Exception occurred")
else:
    print("Hi I am else block")
```

**Output:**

Exception occurred

# Cntd…

```
try:
    #this will throw an exception if the file doesn't exist.
    f = open("file.txt","r")
except IOError:
    print("File not found")
else:
    print("The file opened successfully")
    f.close()
```

**Output:**

File not found

# Common Errors: Type Error & Name Error

```
#Type Error
try:
    a='blue’; b= 3
    c= a+b
except TypeError:
    print('Type error because type mismatch')
except:
    print('error other than TypeError')
else:   print('No error')
# Name Error
try:
    print(k)
except NameError:
    print('Name error')
except:
    print('error other than NameError')
else:   print('No error')
```

# Common Errors: Value Error & Index Error

```
# Value Error
try:
    val=int(input('enter any string:'))
except ValueError:
    print('Value error')
except:
    print('error other than ValueError')
else:   print('No error')
# Index Error
try:
    t= [7,4,36.6,'hi']
    print (t[4])
except IndexError:
    print('Index error')
except: print('error other than IndexError')
else:
    print('No error')
```

# Cntd…

```
try:
   a= 5+ 'c'
except ValueError:
   print('ValueErrors are caught here')
except (NameError, TypeError, ZeroDivisionError): # handle multiple exceptions
   print('NameErrors, TypeErrors and ZeroDivisionErrors are caught here')
except:
   print('any other error’)
```

**Output:**

NameErrors, TypeErrors and ZeroDivisionErrors are caught here

# Cntd…

```
try:
   f = open("test.txt")   # ‘test.txt’ file is not in current directory
finally:
    print('This is finally block')
    f.close()
```

**Output:**

```
This is finally block
NameError: name 'f' is not defined
```

**Note:** If file doesn’t exist and error occurs, after that also finally block will be executed.

# Raising an Exception

- An exception can be raised for user specified conditions using ‘raise’ clause.

```
try:
    age = int(input("Enter the age?"))
    if age<18:
        raise ValueError;
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

**Output:**

```
Enter the age?15
The age is not valid
```

# Cntd..

- Re-writing previous example with some modifications:

```
try:
    age = int(input("Enter the age?"))
    if age<18:
        raise ValueError('Age less than 18');
    else:
        print("the age is valid")
except ValueError as e:
    print("Age not valid: "+str(e))
```

**Output:**

```
Enter the age?14
Age not valid: Age less than 18
```

# Creating User-Defined (Custom) Exception

- The python allows us to create our own exceptions that can be raised from the program and caught using the except clause. Example,

```
class ErrorInCode(Exception):
    def __init__(self, val):
        self.data = val
try:
    raise ErrorInCode(2000)
except ErrorInCode as ae:
    print("Received error:", ae.data)
```

**Output:**

```
Received error: 2000
```

# Assertion

- Assertions in any programming language are the **debugging tools** which help in smooth flow of code.
- Assertions are mainly assumptions and a programmer knows it always must be true. Hence, puts them in between code so that failure of them doesn’t allow the code to execute further.
- In python **assert** keyword helps in achieving this task. This statement simply takes input a boolean condition, which when returns **true** doesn’t return anything, but if it is computed to be **false**, then it raises an AssertionError along with the optional message provided.

# Syntax

- Syntax : assert condition, error_message(optional)
- **Parameters :**
- condition : The boolean condition returning true or false.
- error_message : The optional argument to be printed in console in case of AssertionError
- **Returns :**
- Returns AssertionError, in case the condition evaluates to **false** along with the error message which when provided.

# Cntd…

- Example:

```
a = 4
b = int(input('Enter b: '));
print ("The value of a / b is : ")
assert b != 0, "Divide by 0 error"
print (a / b)
```

**Output:**

```
Enter b: 0
The value of a / b is :
AssertionError: Divide by 0 error
```

# Practical example of assertion

- Assertion has much greater utility in testing and Quality assurance role in any development domain. Different types of assertions are used depending upon the application. Below program only allows the batch with all hot food (i.e. temperature >=27) be dispatched, else rejects whole batch.

- Program:

```
batch = [ 40, 27, 39, 25, 21]          # initializing list of foods temperatures
cut = 27 #Cut off temperature
for i in batch:
    assert i >= 27, "Batch is Rejected"   #Checking valid temperature using assert
    print (str(i) + " is O.K" )
```

**Output:**

```
40 is O.K
27 is O.K
39 is O.K
AssertionError: Batch is Rejected
```

# Abstract Class (Abstract Base Class- ABC)

- A class which contains zero or more abstract methods is called an abstract class.
- An abstract method is a method that has declaration but not has any implementation.
- Abstract classes can not be instantiated and it needs subclasses to provide implementations for those abstract methods.
- While we are designing large functional units we use an abstract class.
- When abstract class has all abstract methods, then behaves like interface.
- There is no concept like interface in python and it is not required also because python supports multiple inheritance and other languages like Java mainly requires interface to provide functionality similar to multiple inheritance.

# Cntd…

- In python by default, it is not able to provide abstract classes, but python comes up with a module which provides the base for defining Abstract Base classes(ABC) and that module name is ABC.
- ABC works by marking methods of the base class as abstract and then registering concrete classes as implementations of the abstract methods.
- A method becomes an abstract by using **decorator @abstractmethod**.
- Forgetting to implement abstract methods in one of the subclasses raises an error.

# Examples of Abstract Class

Example-1:

```
class Polygon:
    def sides(self):  #unimplemented method
        pass
class Triangle(Polygon):
    pass
R = Triangle()
R.sides()
```

**Output: --**

Example-1 is not an example of abstract class because subclass Triangle doesn't implement 'sides' method of parent class, still program doesn't give error and we have not imported module 'abc' also. See Example-2 to understand abstract class.

Example-2:

```
import abc
class Polygon(abc.ABC):
    @abc.abstractmethod
    def sides(self):
        pass
class Triangle(Polygon):
        pass
R = Triangle()
R.sides()
```

**Output:** TypeError: Can't instantiate…

# Find out difference of following programs

Example-3:

```
import abc
class Polygon(abc.ABC):
    @abc.abstractmethod
    def sides(self):
        pass
class Triangle(Polygon):
    def sides(self):
        print("I have 3 sides")
R = Triangle()
R.sides()
```

**Output:** I have 3 sides

Example-4:

```
from abc import *
class Polygon(ABC):
    @abstractmethod
    def sides(self):
        pass
class Triangle(Polygon):
    def sides(self):
        print("I have 3 sides")
R = Triangle()
R.sides()
```

**Output:** I have 3 sides

**Note:** Both programs are example of abstract base class (ABC).