**Unit-IV: Classes and Exceptions**

**Design with Classes**: Classes and Objects, Classes and Functions, Classes and Methods, Working with Instances, Inheritance and Polymorphism. Object-Oriented Programming: Procedural and Object-Oriented Programming, Classes, techniques for Designing Classes.

**Object Oriented Programming**

Python is a multi-paradigm programming language. It supports different programming approaches.

One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

* Attributes (state)
* behavior

Let's take an example:

A parrot is an object, as it has the following properties:

* name, age, color as attributes
* singing, dancing as behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

In Python, the concept of OOP follows some basic principles:

## Class

A class is a blueprint for the object.

We can think of class as a sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, a parrot is an object.

The example for class of parrot can be :

class Parrot:

pass

Here, we use the class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

## Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

obj = Parrot()

Here, obj is an object of class Parrot.

Suppose we have details of parrots. Now, we are going to show how to build the class and objects of parrots.

### Example 1: Creating Class and Object in Python

class Parrot:

# class attribute

species = "bird"

# instance attribute

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# instantiate the Parrot class

blu = Parrot("Blu", 10)

woo = Parrot("Woo", 15)

# access the class attributes

print("Blu is a {}".format(blu.\_\_class\_\_.species))

print("Woo is also a {}".format(woo.\_\_class\_\_.species))

# access the instance attributes

print("{} is {} years old".format( blu.name, blu.age))

print("{} is {} years old".format( woo.name, woo.age))

**Output**

Blu is a bird

Woo is also a bird

Blu is 10 years old

Woo is 15 years old

In the above program, we created a class with the name Parrot. Then, we define attributes. The attributes are a characteristic of an object.

These attributes are defined inside the \_\_init\_\_ method of the class. It is the initializer method that is first run as soon as the object is created.

Then, we create instances of the Parrot class. Here, blu and woo are references (value) to our new objects.

We can access the class attribute using \_\_class\_\_.species. Class attributes are the same for all instances of a class. Similarly, we access the instance attributes using blu.name and blu.age. However, instance attributes are different for every instance of a class.

**Methods**

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

### Example 2 : Creating Methods in Python

class Parrot:

# instance attributes

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

# instance method

def sing(self, song):

return "{} sings {}".format(self.name, song)

def dance(self):

return "{} is now dancing".format(self.name)

# instantiate the object

blu = Parrot("Blu", 10)

# call our instance methods

print(blu.sing("'Happy'"))

print(blu.dance())

**Output**

Blu sings 'Happy'

Blu is now dancing

In the above program, we define two methods i.e sing() and dance(). These are called instance methods because they are called on an instance object i.e blu.

## Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

### Example 3: Use of Inheritance in Python

# parent class

class Bird:

def \_\_init\_\_(self):

print("Bird is ready")

def whoisThis(self):

print("Bird")

def swim(self):

print("Swim faster")

# child class

class Penguin(Bird):

def \_\_init\_\_(self):

# call super() function

super().\_\_init\_\_()

print("Penguin is ready")

def whoisThis(self):

print("Penguin")

def run(self):

print("Run faster")

peggy = Penguin()

peggy.whoisThis()

peggy.swim()

peggy.run()

**Output**

Bird is ready

Penguin is ready

Penguin

Swim faster

Run faster

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from the swim() method.

Again, the child class modified the behavior of the parent class. We can see this from the whoisThis() method. Furthermore, we extend the functions of the parent class, by creating a new run() method.

Additionally, we use the super() function inside the \_\_init\_\_() method. This allows us to run the \_\_init\_\_() method of the parent class inside the child class.

## Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevents data from direct modification which is called encapsulation. In Python, we denote private attributes using underscore as the prefix i.e single \_ or double \_\_.

### Example 4: Data Encapsulation in Python

class Computer:

def \_\_init\_\_(self):

self.\_\_maxprice = 900

def sell(self):

print("Selling Price: {}".format(self.\_\_maxprice))

def setMaxPrice(self, price):

self.\_\_maxprice = price

c = Computer()

c.sell()

# change the price

c.\_\_maxprice = 1000

c.sell()

# using setter function

c.setMaxPrice(1000)

c.sell()

**Output**

Selling Price: 900

Selling Price: 900

Selling Price: 1000

In the above program, we defined a Computer class.

We used \_\_init\_\_() method to store the maximum selling price of Computer. Here, notice the code

c.\_\_maxprice = 1000

Here, we have tried to modify the value of \_\_maxprice outside of the class. However, since \_\_maxprice is a private variable, this modification is not seen on the output.

As shown, to change the value, we have to use a setter function i.e setMaxPrice() which takes price as a parameter.

## Polymorphism

Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

Suppose, we need to color a shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to color any shape. This concept is called Polymorphism.

### Example 5: Using Polymorphism in Python

class Parrot:

def fly(self):

print("Parrot can fly")

def swim(self):

print("Parrot can't swim")

class Penguin:

def fly(self):

print("Penguin can't fly")

def swim(self):

print("Penguin can swim")

# common interface

def flying\_test(bird):

bird.fly()

#instantiate objects

blu = Parrot()

peggy = Penguin()

# passing the object

flying\_test(blu)

flying\_test(peggy)

**Output**

Parrot can fly

Penguin can't fly

In the above program, we defined two classes Parrot and Penguin. Each of them have a common fly() method. However, their functions are different.

To use polymorphism, we created a common interface i.e flying\_test() function that takes any object and calls the object's fly() method. Thus, when we passed the blu and peggy objects in the flying\_test() function, it ran effectively.

**Procedural Programming:**

[Procedural Programming](https://www.geeksforgeeks.org/introduction-of-programming-paradigms/) can be defined as a programming model which is derived from structured programming, based upon the concept of calling procedure. Procedures, also known as routines, subroutines or functions, simply consist of a series of computational steps to be carried out. During a program’s execution, any given procedure might be called at any point, including by other procedures or itself.

**Languages used in Procedural Programming:**

FORTRAN, ALGOL, COBOL,

BASIC, Pascal and C.

**Object Oriented Programming:**

[Object oriented programming](https://www.geeksforgeeks.org/basic-concepts-of-object-oriented-programming-using-c/) can be defined as a programming model which is based upon the concept of objects. Objects contain data in the form of attributes and code in the form of methods. In object oriented programming, computer programs are designed using the concept of objects that interact with real world. Object oriented programming languages are various but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.

**Languages used in Object Oriented Programming:**

Java, C++, C#, Python,

PHP, JavaScript, Ruby, Perl,

Objective-C, Dart, Swift, Scala.

**Difference between Procedural Programming and Object Oriented Programming:**

| Procedural Oriented Programming | Object Oriented Programming |
| --- | --- |
| In procedural programming, program is divided into small parts called ***functions***. | In object oriented programming, program is divided into small parts called ***objects***. |
| Procedural programming follows ***top down approach***. | Object oriented programming follows ***bottom up approach***. |
| There is no access specifier in procedural programming. | Object oriented programming have access specifiers like private, public, protected etc. |
| Adding new data and function is not easy. | Adding new data and function is easy. |
| Procedural programming does not have any proper way for hiding data so it is ***less secure***. | Object oriented programming provides data hiding so it is ***more secure***. |
| In procedural programming, overloading is not possible. | Overloading is possible in object oriented programming. |
| In procedural programming, function is more important than data. | In object oriented programming, data is more important than function. |
| Procedural programming is based on ***unreal world***. | Object oriented programming is based on ***real world***. |
| Examples: C, FORTRAN, Pascal, Basic etc. | Examples: C++, Java, Python, C# etc. |

## Python Objects and Classes

Python is an object-oriented programming language. Unlike procedure-oriented programming, where the main emphasis is on functions, object-oriented programming stresses on objects.

An object is simply a collection of data (variables) and methods (functions) that act on those data. Similarly, a class is a blueprint for that object.

## Defining a Class in Python

Like function definitions begin with the [def](https://www.programiz.com/python-programming/keyword-list#def) keyword in Python, class definitions begin with a [class](https://www.programiz.com/python-programming/keyword-list#class) keyword.

The first string inside the class is called docstring and has a brief description of the class. Although not mandatory, this is highly recommended.

Here is a simple class definition.

class MyNewClass:

'''This is a docstring. I have created a new class'''

pass

A class creates a new local [name space](https://www.programiz.com/python-programming/namespace) where all its attributes are defined. Attributes may be data or functions.

There are also special attributes in it that begins with double underscores \_\_. For example, \_\_doc\_\_ gives us the docstring of that class.

As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

class Person:

"This is a person class"

age = 10

def greet(self):

print('Hello')

# Output: 10

print(Person.age)

# Output: <function Person.greet>

print(Person.greet)

# Output: "This is a person class"

print(Person.\_\_doc\_\_)

**Output**

10

<function Person.greet at 0x7fc78c6e8160>

This is a person class

## Creating an Object in Python

We saw that the class object could be used to access different attributes.

It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a [function](https://www.programiz.com/python-programming/function) call.

>>> harry = Person()

This will create a new object instance named harry. We can access the attributes of objects using the object name prefix.

Attributes may be data or method. Methods of an object are corresponding functions of that class.

## Constructors in Python

Class functions that begin with double underscore \_\_ are called special functions as they have special meaning.

Of one particular interest is the \_\_init\_\_() function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

Constructors are generally used for instantiating an object. The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created. In Python the \_\_init\_\_() method is called the constructor and is always called when an object is created.  
Syntax of constructor declaration :

def \_\_init\_\_(self):

# body of the constructor

**Types of constructors :**

* default constructor: The default constructor is a simple constructor which doesn’t accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.
* parameterized constructor: constructor with parameters is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

**Example of default constructor :**   
 class GeekforGeeks:

    # default constructor

    def \_\_init\_\_(self):

        self.geek = "GeekforGeeks"

    # a method for printing data members

    def print\_Geek(self):

        print(self.geek)

# creating object of the class

obj = GeekforGeeks()

# calling the instance method using the object obj

obj.print\_Geek()

**Output :**

GeekforGeeks

**Example of the** **parameterized constructor :**

|  |
| --- |
| class Addition:      first = 0      second = 0      answer = 0        # parameterized constructor      def \_\_init\_\_(self, f, s):          self.first = f          self.second = s        def display(self):          print("First number = " + str(self.first))          print("Second number = " + str(self.second))          print("Addition of two numbers = " + str(self.answer))        def calculate(self):          self.answer = self.first + self.second    # creating object of the class  # this will invoke parameterized constructor  obj = Addition(1000, 2000)    # perform Addition  obj.calculate()    # display result  obj.display() |

Output :

First number = 1000

Second number = 2000

Addition of two numbers = 3000

**Destructors in Python**

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically.  
The \_\_[del](https://www.geeksforgeeks.org/delattr-del-python/)\_\_() method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.  
Syntax of destructor declaration :

def \_\_del\_\_(self):

# body of destructor

**Note :** A reference to objects is also deleted when the object goes out of reference or when the program ends.

**Example 1 :** Here is the simple example of destructor. By using del keyword we deleted the all references of object ‘obj’, therefore destructor invoked automatically.

# Python program to illustrate destructor

class Employee:

# Initializing

def \_\_init\_\_(self):

print('Employee created.')

# Deleting (Calling destructor)

def \_\_del\_\_(self):

print('Destructor called, Employee deleted.')

obj = Employee()

del obj

Output:

Employee created.

Destructor called, Employee deleted.

# Python Inheritance

Inheritance is the capability of one class to derive or inherit the properties from 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.

Below is a simple example of inheritance in Python

**Parent class** is the class being inherited from, also called base class.

**Child class** is the class that inherits from another class, also called derived class.

# A Python program to demonstrate inheritance

# Base or Super class. Note object in bracket.

# (Generally, object is made ancestor of all classes)

# In Python 3.x "class Person" is

# equivalent to "class Person(object)"

class Person(object):

    # Constructor

    def \_\_init\_\_(self, name):

        self.name = name

    # To get name

    def getName(self):

        return self.name

    # To check if this person is an employee

    def isEmployee(self):

        return False

# Inherited or Subclass (Note Person in bracket)

class Employee(Person):

    # Here we return true

    def isEmployee(self):

        return True

# Driver code

emp = Person("John")  # An Object of Person

print(emp.getName(), emp.isEmployee())

emp = Employee("Alex") # An Object of Employee

print(emp.getName(), emp.isEmployee())

**Output:**

John False

Alex True

**What is object class?**   
Like [Java Object class](https://www.geeksforgeeks.org/object-class-in-java/), in Python (from version 3.x), object is root of all classes.   
In Python 3.x, “class Test(object)” and “class Test” are same.   
In Python 2.x, “class Test(object)” creates a class with object as parent (called new style class) and “class Test” creates old style class (without object parent)

**Subclassing (Calling constructor of parent class)**   
A child class needs to identify which class is its parent class. This can be done by mentioning the parent class name in the definition of the child class.   
Eg: class **subclass\_name (superclass\_name)**:

# Python code to demonstrate how parent constructors

# are called.

# parent class

class Person( object ):

        # \_\_init\_\_ is known as the constructor

        def \_\_init\_\_(self, name, idnumber):

                self.name = name

                self.idnumber = idnumber

        def display(self):

                print(self.name)

                print(self.idnumber)

# child class

class Employee( Person ):

        def \_\_init\_\_(self, name, idnumber, salary, post):

                self.salary = salary

                self.post = post

                # invoking the \_\_init\_\_ of the parent class

                Person.\_\_init\_\_(self, name, idnumber)

# creation of an object variable or an instance

a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using its instance

a.display()

**Output:**

Rahul

886012

Example 2:

# Python program to demonstrate error if we

# forget to invoke \_\_init\_\_() of the parent.

class A:

      def \_\_init\_\_(self, n = 'Rahul'):

              self.name = n

class B(A):

      def \_\_init\_\_(self, roll):

              self.roll = roll

object = B(23)

print (object.name)

**Output :**

Traceback (most recent call last):

File "/home/de4570cca20263ac2c4149f435dba22c.py", line 12, in

print (object.name)

AttributeError: 'B' object has no attribute 'name'

## Add the \_\_init\_\_() Function

So far we have created a child class that inherits the properties and methods from its parent.

We want to add the \_\_init\_\_() function to the child class (instead of the pass keyword).

**Note:** The \_\_init\_\_() function is called automatically every time the class is being used to create a new object.

To keep the inheritance of the parent's \_\_init\_\_() function, add a call to the parent's \_\_init\_\_() function:

### Example

class Student(Person):  
  def \_\_init\_\_(self, fname, lname):  
    Person.\_\_init\_\_(self, fname, lname)

## Use the super () Function

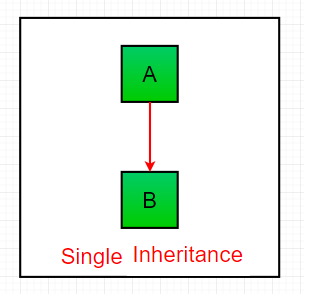
Python also has a super() function that will make the child class inherit all the methods and properties from its parent:

### Example

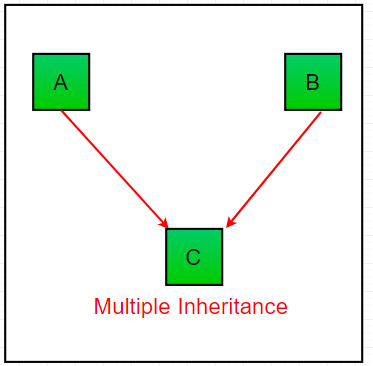
class Student(Person):  
  def \_\_init\_\_(self, fname, lname):  
    super().\_\_init\_\_(fname, lname)

**Different forms of Inheritance:**

**1. Single inheritance**: When a child class inherits from only one parent class, it is called single inheritance. We saw an example above.



**2. Multiple inheritance**: When a child class inherits from multiple parent classes, it is called multiple inheritance.   
Unlike Java and like C++, Python supports multiple inheritance. We specify all parent classes as a comma-separated list in the bracket.



|  |
| --- |
| # Python example to show the working of multiple  # inheritance  class Base1(object):      def \_\_init\_\_(self):          self.str1 = "AIML"          print("Base1")    class Base2(object):      def \_\_init\_\_(self):          self.str2 = "DS"          print("Base2")    class Derived(Base1, Base2):      def \_\_init\_\_(self):            # Calling constructors of Base1          # and Base2 classes          Base1.\_\_init\_\_(self)          Base2.\_\_init\_\_(self)          print("Derived")        def printStrs(self):          print(self.str1, self.str2)      ob = Derived()  ob.printStrs() |

**Output:**

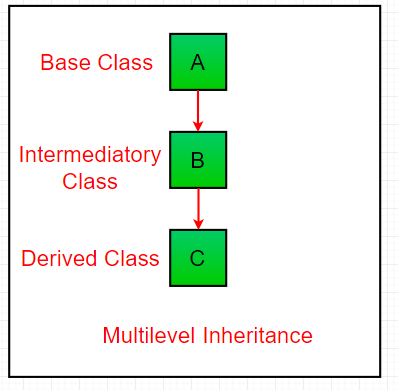
Base1

Base2

Derived

AIML DS

1. **Multilevel Inheritance**   
   In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and grandfather.



|  |
| --- |
| # Python program to demonstrate  # multilevel inheritance    # Base class  class Grandfather:        def \_\_init\_\_(self, grandfathername):          self.grandfathername = grandfathername    # Intermediate class  class Father(Grandfather):      def \_\_init\_\_(self, fathername, grandfathername):          self.fathername = fathername            # invoking constructor of Grandfather class          Grandfather.\_\_init\_\_(self, grandfathername)    # Derived class  class Son(Father):      def \_\_init\_\_(self,sonname, fathername, grandfathername):          self.sonname = sonname            # invoking constructor of Father class          Father.\_\_init\_\_(self, fathername, grandfathername)        def print\_name(self):          print('Grandfather name :', self.grandfathername)          print("Father name :", self.fathername)          print("Son name :", self.sonname)    #  Driver code  s1 = Son('Prince', 'Rampal', 'Lal mani')  print(s1.grandfathername)  s1.print\_name() |

**Output:**

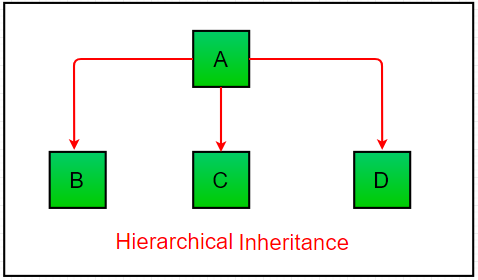
Lal mani

Grandfather name : Lal mani

Father name : Rampal

Son name : Prince

1. **Hierarchical Inheritance:** When more than one derived classes are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.



|  |
| --- |
| # Python program to demonstrate  # Hierarchical inheritance      # Base class  class Parent:        def func1(self):            print("This function is in parent class.")    # Derived class1  class Child1(Parent):        def func2(self):            print("This function is in child 1.")    # Derivied class2  class Child2(Parent):        def func3(self):            print("This function is in child 2.")    # Driver's code  object1 = Child1()  object2 = Child2()  object1.func1()  object1.func2()  object2.func1()  object2.func3() |

**Output:**

This function is in parent class.

This function is in child 1.

This function is in parent class.

This function is in child 2.

**Hybrid Inheritance:** Inheritance consisting of multiple types of inheritance is called hybrid inheritance.

|  |
| --- |
| # Python program to demonstrate  # hybrid inheritance      class School:       def func1(self):           print("This function is in school.")    class Student1(School):       def func2(self):           print("This function is in student 1. ")    class Student2(School):       def func3(self):           print("This function is in student 2.")    class Student3(Student1, School):       def func4(self):           print("This function is in student 3.")    # Driver's code  object = Student3()  object.func1()  object.func2() |

**Output:**

This function is in school.

This function is in student 1.

**Polymorphism in Python**

**What is Polymorphism:** The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.  
**Example of inbuilt polymorphic functions :**

# Python program to demonstrate in-built poly-morphic functions

# len() being used for a string

print(len("geeks"))

# len() being used for a list

print(len([10, 20, 30]))

**Output:**

5

3

**Examples of user-defined polymorphic functions :**

# A simple Python function to demonstrate

# Polymorphism

def add(x, y):

    return x + y

def add(x, y, z):

    return x + y + z

# Driver code

print(add(2, 3))

print(add(2, 3, 4))

**Output:**

5

9

**Polymorphism with class methods:**   
The below code shows how Python can use two different class types, in the same way. We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is. We assume that these methods actually exist in each class.

class India():

    def capital(self):

        print("New Delhi is the capital of India.")

    def language(self):

        print("Hindi is the most widely spoken language of India.")

    def type(self):

        print("India is a developing country.")

class USA():

    def capital(self):

        print("Washington, D.C. is the capital of USA.")

    def language(self):

        print("English is the primary language of USA.")

    def type(self):

        print("USA is a developed country.")

obj\_ind = India()

obj\_usa = USA()

for country in (obj\_ind, obj\_usa):

    country.capital()

    country.language()

    country.type()

**Output:**

New Delhi is the capital of India.

Hindi is the most widely spoken language of India.

India is a developing country.

Washington, D.C. is the capital of USA.

English is the primary language of USA.

USA is a developed country.

**Polymorphism with Inheritance:**   
In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class doesn’t quite fit the child class. In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as **Method Overriding**.   
 class Bird:

  def intro(self):

    print("There are many types of birds.")

  def flight(self):

    print("Most of the birds can fly but some cannot.")

class sparrow(Bird):

  def flight(self):

    print("Sparrows can fly.")

class ostrich(Bird):

  def flight(self):

    print("Ostriches cannot fly.")

obj\_bird = Bird()

obj\_spr = sparrow()

obj\_ost = ostrich()

obj\_bird.intro()

obj\_bird.flight()

obj\_spr.intro()

obj\_spr.flight()

obj\_ost.intro()

obj\_ost.flight()

**OUTPUT**

There are many types of birds.

Most of the birds can fly but some cannot.

There are many types of birds.

Sparrows can fly.

There are many types of birds.

Ostriches cannot fly.

# Operator Overloading in Python

Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class. You might have noticed that the same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*.

# Python program to show use of

# + operator for different purposes.

print(1 + 2)

# concatenate two strings

print("AIML"+"DS")

# Product two numbers

print(3 \* 4)

# Repeat the String

print("CMR"\*4)

When we use an operator on user defined data types then automatically a special function or magic function associated with that operator is invoked. Changing the behavior of operator is as simple as changing the behavior of method or function. You define methods in your class and operators work according to that behavior defined in methods. When we use + operator, the magic method \_\_add\_\_ is automatically invoked in which the operation for + operator is defined. There by changing this magic method’s code, we can give extra meaning to the + operator.

#### Python magic methods or special functions for operator overloading

#### [Binary Operators](https://www.geeksforgeeks.org/basic-operators-python/):

|  |  |
| --- | --- |
| Operator | Magic Method |
| **+** | \_\_add\_\_(self, other) |
| **–** | \_\_sub\_\_(self, other) |
| **\*** | \_\_mul\_\_(self, other) |
| **/** | \_\_truediv\_\_(self, other) |
| **//** | \_\_floordiv\_\_(self, other) |
| **%** | \_\_mod\_\_(self, other) |
| **\*\*** | \_\_pow\_\_(self, other) |
| >> | \_\_rshift\_\_(self, other) |
| << | \_\_lshift\_\_(self, other) |
| & | \_\_and\_\_(self, other) |
| | | \_\_or\_\_(self, other) |
| ^ | \_\_xor\_\_(self, other) Comparison Operators :  |  |  | | --- | --- | | Operator | Magic 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) | | **>>=** | \_\_IRSHIFT\_\_(SELF, OTHER) | | **<<=** | \_\_ILSHIFT\_\_(SELF, OTHER) | | **&=** | \_\_IAND\_\_(SELF, OTHER) | | **|=** | \_\_IOR\_\_(SELF, OTHER) | | **^=** | \_\_IXOR\_\_(SELF, OTHER) |  Unary Operators :  |  |  | | --- | --- | | Operator | Magic Method | | **–** | \_\_NEG\_\_(SELF, OTHER) | | **+** | \_\_POS\_\_(SELF, OTHER) | | **~** | \_\_INVERT\_\_(SELF, OTHER) | |

**Example 2**

# Python Program illustrate how

# to overload an binary + operator

class A:

    def \_\_init\_\_(self, a):

        self.a = a

    # adding two objects

    def \_\_add\_\_(self, o):

        return self.a + o.a

ob1 = A(1)

ob2 = A(2)

ob3 = A("AIML")

ob4 = A("DS")

print(ob1 + ob2)

print(ob3 + ob4)

**Output :** 

3

AIMLDS

**Example 3**

# Python program to overload

# a comparison operators

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

**Class Variable:**A class variable is nothing but a variable that is defined outside the constructor. A class variable is also called as a **static variable**.

**Instance variable − A variable that is defined inside a method and belongs only to the current instance of a class**.

**Accessor(Getters):** If you want to fetch the value from an instance variable we call them accessors.

**Mutator(Setters):**If you want to modify the value we call them mutators.

# Types Of Methods In Python

Generally, there are three types of methods in Python:

1. Instance Methods.
2. Class Methods
3. Static Methods

## 1. Instance Method

This is a very basic and easy method that we use regularly when we create [classes in python](https://www.studytonight.com/python/class-in-python). If we want to print an instance variable or instance method we must create an object of that required class.

If we are using self as a function parameter or in front of a variable, that is nothing but the calling instance itself.

As we are working with instance variables we use self keyword.

**Note:**Instance variables are used with instance methods.

# Instance Method Example in Python

class Student:

def \_\_init\_\_(self, a, b):

self.a = a

self.b = b

def avg(self):

return (self.a + self.b) / 2

s1 = Student(10, 20)

print( s1.avg() )

**Output:**

15.0

In the above program, a and b are instance variables and these get initialized when we create an object for the Student class. If we want to call avg() function which is an instance method, we must create an object for the class.

If we clearly look at the program, the self keyword is used so that we can easily say that those are instance variables and methods.

2. Class Method

classsmethod() function returns a class method as output for the given function.

Here is the syntax for it:

classmethod(function)

The classmethod() method takes only a function as an input parameter and converts that into a class method.

There are two ways to create class methods in python:

1. Using classmethod(function)
2. Using @classmethod annotation

A class method can be called either using the class (such as C.f()) or using an instance (such as C().f()). The instance is ignored except for its class. If a class method is called from a derived class, the derived class object is passed as the implied first argument.

As we are working with ClassMethod we use the cls keyword. Class variables are used with class methods.

Look at the code below.

# Class Method Implementation in python

class Student:

name = 'Student'

def \_\_init\_\_(self, a, b):

self.a = a

self.b = b

@classmethod

def info(cls):

return cls.name

print(Student.info())

Copy

**Output:**

Student

In the above example, name is a class variable. If we want to create a class method we must use @classmethod decorator and cls as a parameter for that function.

3. Static Method

A static method can be called without an object for that class, using the class name directly. If you want to do something extra with a class we use static methods.

For example, If you want to print factorial of a number then we don't need to use class variables or instance variables to print the factorial of a number. We just simply pass a number to the static method that we have created and it returns the factorial.

Look at the below code

# Static Method Implementation in python

class Student:

name = 'Student'

def \_\_init\_\_(self, a, b):

self.a = a

self.b = b

@staticmethod

def info():

return "This is a student class"

print(Student.info())

Copy

**Output**

This a student class