**PYTHON**

**UNIT-III**

**Python File Handling**

Till now, we were taking the input from the console and writing it back to the console to interact with the user.

Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again.

However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling.

file handling in python including, creating a file, opening a file, closing a file, writing and appending the file, etc.

**Opening a file**

Python provides the open() function which accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

The syntax to use the open() function is given below.

**file object = open(<file-name>, <access-mode>, <buffering>)**

The files can be accessed using various modes like read, write, or append.

The following are the details about the access mode to open a file.

|  |  |  |
| --- | --- | --- |
| **SN** | **Access mode** | **Description** |
| 1 | r | It opens the file to read-only. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed. |
| 2 | rb | It opens the file to read only in binary format. The file pointer exists at the beginning of the file. |
| 3 | r+ | It opens the file to read and write both. The file pointer exists at the beginning of the file. |
| 4 | rb+ | It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file. |
| 5 | w | It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file. |
| 6 | wb | It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file. |
| 7 | w+ | It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new file if no file exists. The file pointer exists at the beginning of the file. |
| 8 | wb+ | It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file. |
| 9 | a | It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a new file if no file exists with the same name. |
| 10 | ab | It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name. |
| 11 | a+ | It opens a file to append and read both. The file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name. |
| 12 | ab+ | It opens a file to append and read both in binary format. The file pointer remains at the end of the file. |

**open() :** it is to open a file.

Example: To open a file named "file.txt" (stored in the same directory) in read mode.

#opens the file file.txt in read mode

fileptr = open("file1.txt","r")

if fileptr:

    print("file is opened successfully")

**Output:**

<class '\_io.TextIOWrapper'>

file is opened successfully

**close() :** it is used to close an opened file.

it is good practice to close the file once all the operations are done.

The syntax to use the close() method is given below.

Ex: fileobject.close()

**Example**

# opens the file file.txt in read mode

fileptr = open("file.txt","r")

if fileptr:

    print("file is opened successfully")

#closes the opened file

fileptr.close()

**read() :** it is used to read data content of a file. It can also read data in binary format.

Syntax:

fileobj.read(<count>)

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Consider the following example.

**Example**

#open the file.txt in read mode. causes error if no such file exists.

fileptr = open("file.txt","r");

#stores all the data of the file into the variable content

content = fileptr.read(9);

# prints the type of the data stored in the file

print(type(content))

#prints the content of the file

print(content)

#closes the opened file

fileptr.close()

**Output:**

<class 'str'>

Hi, I am

**Read Lines of the file :**

The readline() method reads the lines of the file from the beginning, i.e., if we use the readline() method two times, then we can get the first two lines of the file.

**Example**

#open the file.txt in read mode. causes error if no such file exists.

fileptr = open("file.txt","r");

#stores all the data of the file into the variable content

content = fileptr.readline();

# prints the type of the data stored in the file

print(type(content))

#prints the content of the file

print(content)

#closes the opened file

fileptr.close()

**Output:**

<class 'str'>

Hi, I am the file and being used as

**Looping through the file :**

By looping through the lines of the file, we can read the whole file.

**Example**

#open the file.txt in read mode. causes an error if no such file exists.

fileptr = open("file.txt","r");

#running a for loop

for i in fileptr:

    print(i) # i contains each line of the file

**Output:**

Hi, I am the file and being used as

an example to read a

file in python.

**Writing the file**

To write some text to a file, we need to open the file using the open method with one of the following access modes.

**a:** It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

**w:** It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

Consider the following example.

Example 1

#open the file.txt in append mode. Creates a new file if no such file exists.

fileptr = open("file.txt","a");

#appending the content to the file

fileptr.write("Python is the modern day language. It makes things so simple.")

#closing the opened file

fileptr.close();

Example 2

#open the file.txt in write mode.

fileptr = open("file.txt","w");

#overwriting the content of the file

fileptr.write("Python is the modern day language. It makes things so simple.")

#closing the opened file

fileptr.close();

Now, we can check that all the previously written content of the file is overwritten with the new text

**Creating a new file**

The new file can be created by using one of the following access modes with the function open()

**x:** it creates a new file with the specified name. It causes an error a file exists with the same name.

**a:** It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

**w:** It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Consider the following example.

Example

#open the file.txt in read mode. causes error if no such file exists.

fileptr = open("file2.txt","x");

print(fileptr)

if fileptr:

    print("File created successfully");

Output:

File created successfully

**Using with statement with files**

The with statement is useful in the case of manipulating the files.

Syntax:

with open(<file name>, <access mode>) as <file-pointer>:

It is always suggestible to use the with statement in the case of files because, It doesn't let the file to be corrupted.

Example:

with open("file.txt",'r') as f:

    content = f.read();

    print(content)

**tell():**

method which is used to print the byte number at which the file pointer exists. Consider the following example.

**Example**

# open the file file2.txt in read mode

fileptr = open("file2.txt","r")

#initially the filepointer is at 0

print("The filepointer is at byte :",fileptr.tell())

#reading the content of the file

content = fileptr.read();

#after the read operation file pointer modifies. tell() returns the location of

the fileptr.

print("After reading, the filepointer is at:",fileptr.tell())

**Output:**

The filepointer is at byte : 0

After reading, the filepointer is at 26

seek(): This method which enables us to modify the file pointer position externally.

The seek() method accepts two parameters:

**offset:** It refers to the new position of the file pointer within the file.

**from:** It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. If it is set to 1, the current position of the file pointer is used as the reference position. If it is set to 2, the end of the file pointer is used as the reference position.

Consider the following example.

**Example**

# open the file file2.txt in read mode

fileptr = open("file2.txt","r")

#initially the filepointer is at 0

print("The filepointer is at byte :",fileptr.tell())

#changing the file pointer location to 10.

fileptr.seek(10);

#tell() returns the location of the fileptr.

print("After reading, the filepointer is at:",fileptr.tell())

**Output:**

The filepointer is at byte : 0

After reading, the filepointer is at 10

**Exercises:**

**1.** Write a Python program to read an entire text file.

def file\_read(fname):

txt = open(fname)

print(txt.read())

file\_read('test.txt')

Program1: to read a sample file from the present working directory by default:

fileptr = open(r"c:\\users\\mural\\file1.txt","r")

if fileptr:

print("file is opened successfully")

content = fileptr.read();

print(type(content))

print(content)

fileptr.close()

o/p:

file is opened successfully

<class 'str'>

hello this is a sample file

this is 2nd line

this is 3rd line or sentence

program2: to open a file and reading line by line and then display it.

fileptr = open("file1.txt","r");

content = fileptr.readline();

print(content)

content = fileptr.readline();

print(content)

content = fileptr.readline();

print(content)

#closes the opened file

fileptr.close()

o/p:

hello this is a sample file

this is 2nd line

this is 3rd line or sentence

Program3: reading all the content using loop control:

fileptr = open("file1.txt","r");

#running a for loop

for i in fileptr:

print(i)

else:

print("EOF")

o/p:

hello this is a sample file

this is 2nd line

this is 3rd line or sentence

EOF

Program4: for appending more content into file1.txt

fileptr = open("file1.txt","a");

#appending the content to the file

fileptr.write("Python is the modern day language. It makes things so simple.")

#closing the opened file

fileptr.close();

fileptr = open("file1.txt","r");

#running a for loop

for i in fileptr:

print(i)

else:

print("EOF")

fileptr.close();

program5: create a new empty file, show error if file already exist.

fileptr = open("file2.txt","x");

print(fileptr)

if fileptr:

    print("File created successfully");

Output:

File created successfully

Program6: display the content of a file using with statement.

with open("file1.txt",'r') as f:

content = f.read();

print(content)

o/p:

hello this is a sample file

this is 2nd line

this is 3rd line or sentencePython is the modern day language. It makes things so simple.

Program7: write a program to input a sample paragraph from user.

fileptr = open("file2.txt","a");

fileptr.write(input("enter a sentence"))

fileptr.close();

with open("file2.txt",'r') as f:

content = f.read();

print(content)

fileptr.close()

o/p:

enter a sentencehi hi hi

345asdhello friendshello89374hello friendhi hi hi

(OR)

fileptr = open("file2.txt","a");

fileptr.write(input("enter a sentence"))

fileptr.seek(0) # it rewinds the pointer to the beginning

with open("file2.txt",'r') as f:

content = f.read();

print(content)

fileptr.close()

program8: write a program to copy the content of file1.txt into file3.txt

fp1 = open("file1.txt","r");

fp2=open("file2.txt","w+");

x=fp1.read()

fp2.write(x)

fp1.seek(0)

print("content in file1 is: ")

c1=fp1.read()

print(c1)

print("content in file2 is: ")

fp2.seek(0)

c2=fp2.read()

print(c2)

fp1.close()

fp2.close()

o/p:

content in file1 is:

hi friend

content in file2 is:

hi friend

program9: input a sequence of numbers into a file.

fp1=open("num1.txt","w+")

fp1.write(input("enter numbers"))

fp1.seek(0)

x=fp1.read()

print(x)

o/p:

enter numbers10 20 30

10 20 30

Program10: write a program to input a sequence of number and find their sum:

fp1=open("num1.txt","w+")

fp1.write(input("enter numbers"))

fp1.seek(0)

x=0

st=fp1.read()

list1=st.split()

for i in list1:

x=x+int(i)

print(x)

o/p:

enter numbers10 20 30

60

Program11: enter a paragraph in to a file. Find how many palindromes exist within the file and then display them.

fp1=open("sample.txt","w+")

fp1.write(input("enter a paragraph"))

fp1.seek(0)

x=fp1.read()

list1=x.split()

revlist=list(map(lambda x:x[::-1], list1))

print(list1)

print(revlist)

count=0

print("the palindromes are: ")

for x in range(len(list1)):

if list1[x]==revlist[x]:

count=count+1

print(list1[x])

print("total no of palindromes ",count)

o/p:

enter a paragraphhello madam do you know malayalam

['hello', 'madam', 'do', 'you', 'know', 'malayalam']

['olleh', 'madam', 'od', 'uoy', 'wonk', 'malayalam']

the palindromes are:

madam

malayalam

total no of palindromes 2

**The file related methods**

The file object provides the following methods to manipulate the files on various operating systems.

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1 | file.close() | It closes the opened file. The file once closed, it can't be read or write any more. |
| 2 | File.flush() | It flushes the internal buffer. |
| 3 | File.next() | It returns the next line from the file. |
| 4 | File.read([size]) | It reads the file for the specified size. |
| 5 | File.readline([size]) | It reads one line from the file and places the file pointer to the beginning of the new line. |
| 6 | File.readlines([sizehint]) | It returns a list containing all the lines of the file. It reads the file until the EOF occurs using readline() function. |
| 7 | File.seek(offset[,from) | It modifies the position of the file pointer to a specified offset with the specified reference. |
| 8 | File.tell() | It returns the current position of the file pointer within the file. |
| 9 | File.write(str) | It writes the specified string to a file |
| 10 | File.writelines(seq) | It writes a sequence of the strings to a file. |
|  |  |  |
|  |  |  |
|  |  |  |

**Python os module**

The os module provides us the functions that are involved in file processing operations like renaming, deleting, etc.

Let's look at some of the os module functions.

**Renaming the file**

The os module provides us the rename() method which is used to rename the specified file to a new name. The syntax to use the rename() method is given below.

1. rename(?current-name?, ?new-name?)

Example

import os;

#rename file2.txt to file3.txt

os.rename("file2.txt","file3.txt")

**Removing the file**

The os module provides us the remove() method which is used to remove the specified file. The syntax to use the remove() method is given below.

1. remove(?file-name?)

Example

import os;

#deleting the file named file3.txt

os.remove("file3.txt")

**Creating the new directory**

The mkdir() method is used to create the directories in the current working directory. The syntax to create the new directory is given below.

1. mkdir(?directory name?)

Example

import os;

#creating a new directory with the name new

os.mkdir("new")

**Changing the current working directory**

The chdir() method is used to change the current working directory to a specified directory.

The syntax to use the chdir() method is given below.

1. chdir("new-directory")

Example

import os;

#changing the current working directory to new

os.chdir("new")

**Deleting directory**

The rmdir() method is used to delete the specified directory.

The syntax to use the rmdir() method is given below.

1. os.rmdir(?directory name?)

Example

import os;

#removing the new directory

os.rmdir("new")

**Python IDEs**

IDE stands for Integrated Development Environment is defined as a coding tool that helps to automate the process of editing, compiling, testing, etc.

There are some Python IDEs which are as follows:

[PyCharm](https://www.javatpoint.com/python-ides#PyCharm)

[Spyder](https://www.javatpoint.com/python-ides#Spyder)

[PyDev](https://www.javatpoint.com/python-ides#PyDev)

[Atom](https://www.javatpoint.com/python-ides#Atom)

[Wing](https://www.javatpoint.com/python-ides#Wing)

[Jupyter Notebook](https://www.javatpoint.com/python-ides#JupyterNotebook)

[Thonny](https://www.javatpoint.com/python-ides#Thonny)

[Rodeo](https://www.javatpoint.com/python-ides#Rodeo)

[Microsoft Visual Studio](https://www.javatpoint.com/python-ides#MicrosoftVisualStudio)

[Eric](https://www.javatpoint.com/python-ides#Eric)

**Object Oriented Programming:**

It allows to create their own objects that have methods and attributes.

The commonly repeated tasks and objects can be defined with OOP to create code that is more usable.

class: the basic way to define an object is using class keyword.

Major principles of OOP system are:

1. object
2. class
3. method
4. inheritance
5. polymorphism
6. data abstraction
7. encapsulation

**class:** A class is a blue print for the object.

It is a virtual entity. It comes into existence when its object is created.

Syntax:

class classname:

#statements or members

Ex1:Create a class named MyClass, with a property named x:

class MyClass:  
 x = 5

**Object:** It is an instance of a class

**Syntax:**

**Object name=<class name> (<arguments>)**

Ex1: Now we can use the class named myClass to create objects.

Create an object named p1, and print the value of x:

p1 = MyClass()  
print(p1.x)

ex1:

class myclass:

x=25

obj=myclass()

print(obj.x)

ex2:

class emp:

empid=1

ename="alok"

def display(self):

print(self.empid,self.ename)

obj=emp()

obj.display()

obj.empid=2

obj.ename="sam"

obj.display()

o/p:

1 alok

2 sam

Ex3:

class emp:

id=1

name="abc"

def display(self):

print(self.id, self.name)

x=emp()

x.display()

o/p:

1 abc

\* Here x is an object which allows to call a function. Such as x.display(). When an object calls its method the object itself is passed as the 1st argument to it. So, x.display() is converted as:

emp.display(x)

So, for this reason the 1st argument of the function in the class must be the object itself referred by self-variable.

Ex4:

class lib:

bookid=1

bookname="a"

bookauthor="b"

bookcost=1.1

bookqty=1

def accept(self):

bookid=int(input("enter book id"))

self.bookname=input("enter book name")

self.bookauthor=input("enter book author")

self.bookcost=float(input("enter book cost"))

self.bookqty=int(input("enter quantity"))

def display(self):

print("the book details are\n")

print(self.bookid,self.bookname,self.bookauthor, self.bookcost,self.bookqty)

obj=lib()

obj.accept()

obj.display()

ex5:wap to create a class having member data a,b,c and member functions: accept(), sum() and display()

class myclass:

a=0

b=0

c=0

result=0

def accept(self, x,y,z):

self.a=x

self.b=y

self.c=z

def sum(self):

self.result=self.a+self.b+self.c

def display(self):

print("sum is",self.result)

obj=myclass()

n1=int(input("enter 1st no"))

n2=int(input("enter 2nd no"))

n3=int(input("enter 3rd no"))

obj.accept(n1,n2,n3)

obj.sum()

obj.display()

**self Parameter**

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class:

Example:

class person:  
  def input(obj, name, age):  
    obj.name = name  
    obj.age = age  
  
  def myfunc(abc):  
    print("Hello my name is " + abc.name)

p1=person()

p1.input(“murali”,32)  
p1.myfunc()

when a function is called using the object, actually it is passed as the 1st argument to the function.

Ex:

p1.myfunc()

is modified by interpreter as:

person.myfunc(p1)

So, for this reason the 1st argument of the function in class must be the object itself referred by self variable.

Example2: input two numbers and perform addition

class A:

def input(self, x,y):

self.a=x

self.b=y

def add(self):

return self.a+self.b;

def display(self):

print(self.a, self.b);

obj=A()

t1=int(input("enter 1st no"))

t2=int(input("enter 2nd no"))

obj.input(t1,t2) *or it means A.input(obj,t1,t2)*

print("addition is",obj.add())

o/p:

enter 1st no12

enter 2nd no13

addition is 25

example3:

write a program to create a class having members for product no, name, cost and quantity, total amount

methods: input(), calculate(), display()

calculate() is to find total amount=cost\*quantity

solution:

class product:

pno=0

pname=" "

cost=qty=amt=0

def input(self,p,q,r,name):

self.pno=p

self.cost=q

self.qty=r

self.pname=name

def calc(self):

self.amt=self.cost\*self.qty

def display(self):

print("product no is :",self.pno)

print("product name is :", self.pname)

print("product cost is : ",self.cost)

print("product quantity is :", self.qty)

print("product amount is :",self.amt)

ob=product()

p=int(input("product no"))

q=int(input("product quantity"))

r=int(input("product cost"))

name=input("product name")

ob.input(p,q,r,name)

ob.calc() # product.calc(ob)

ob.display() # product.display(ob)

o/p:

product no1

product quantity25

product cost10

product namelux

product no is : 1

product name is : lux

product cost is : 25

product quantity is : 10

product amount is : 250

**python constructor:**

A constructor is a special type of method which is used to initialize the object members of the class.

**creating the constructor in python:**

In python the method \_ \_init\_ \_ simulates the constructor of the class. This method is called automatically when the class is initiated.

So, the \_\_init\_\_ function used to initialize the class attributes. So, every class should have a constructor.

Ex:

class emp:

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

self.id=id

self.name=name

def display(self):

print(self.id, self.name)

emp1=emp(“john”,101)

emp2=emp(“David”,102)

emp1.display()

emp2.display()

o/p:

101 john

102 david

Constructors can be of two types:

1. non-parameterized constructor
2. parameterized constructor

**Non-parameterized constructor:**

A constructor invoked without parameters during declaring an object.

Example1:

class student:

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

print(“it is not parametrised constructor”)

def show(self, name):

print(“hello”,name)

s1=student()

s1.show(“john”)

o/p:

This is not a parametrized constructor

Hello John

Example2:

Counting the no. of objects of a class.

class student:

count=0

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

student.count=student.count+1

s1.student()

s2.student()

s3.student()

print(“no. of students are “,student.count)

o/p:

no. of students are 3

**Parameterized constructor:**

The constructor needs parameters to be passed when it is invoked.

Example1:

Create a class named Person, use the \_\_init\_\_() function to assign values for name and age:

class Person:  
  def \_\_init\_\_(self, name, age):  
    self.name = name  
    self.age = age  
  
  def myfunc(self):  
    print("Hello my name is " + self.name)  
  
p1 = Person("murali", 32)  
p1.myfunc()

example2: write a program to store N persons details using class concept.

class student:

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

self.name=name

self.age=age

# def display(self):

# print(self.name,self.age)

list1=[]

for i in range(1,4):

n=input("enter name")

a=int(input("enter age"))

# obj=student(n,a)

list1.append(student(n,a))

for obj in list1:

print(obj.name,obj.age)

o/p:

enter nameaa

enter age1

enter namebb

enter age2

enter namecc

enter age3

aa 1

bb 2

cc 3

**python built-in class functions:**

They are:

getattr(obj,name,default)

It is used to access the attribute of the object.  
setattr(obj,name,value)

it is used to set a particular value to the specific attribute of an object. If attribute does not exist, then it would create.

delattr(obj,name)

it is used to delete a specific attribute.

hasattr(obj,name)

it returns TRUE if the object contains some specific attribute.

Example1:

class student:

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

self.name=name

self.id=id

self.age=age

s=student("john",101,22)

print(getattr(s,"id"))

print(getattr(s,"name"))  
o/p:

101

john

example2:

class student:

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

self.name=name

self.id=id

self.age=age

s=student("alok",101,21)

setattr(s,"age",25)

print(getattr(s,"name"))

print(getattr(s,"age"))

o/p:

alok

25

Example3:

class student:

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

self.name=name

self.id=id

self.age=age

s=student("alok",101,21)

print("age attribute exist?",hasattr(s,"age"))

delattr(s,"age")

print("age attribute exist?",hasattr(s,"age"))

o/p:

age attribute exist? True

age attribute exist? False

**built-in class attributes:**

along with other attributes python also contains some built in class attributes which provides information about the class. They are accessed using objects.

They are:

\_ \_dict\_ \_

It is dictionary containing the class’s namespace

\_ \_doc\_ \_

It contains a string which have the class document

\_ \_name\_ \_

It is used to access the class name.

\_ \_module\_ \_

It is used to access the module in which the class is defined.

\_ \_bases\_ \_

It contains a tuple including all base classes. It is possibly an empty tuple.

Example1:

class student:

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

self.name=name

self.age=age

def display(self):

print("name %s and age %d"%(self.name,self.age))

s=student("john",21)

print(s.\_ \_dict\_ \_)

print(s.\_ \_doc\_ \_)

print(s.\_ \_module\_ \_)

o/p:

{'name': 'john', 'age': 21}

None

\_ \_main\_ \_

# **\_ \_name\_ \_ (A Special variable) in Python**

Since there is no main() function in Python, when the command to run a python program is given to the interpreter, the code that is at level 0 indentation is to be executed. However, before doing that, it will define a few special variables. \_\_name\_\_ is one such special variable. If the source file is executed as the main program, the interpreter sets the \_\_name\_\_ variable to have a value “\_\_main\_\_”. If this file is being imported from another module, \_\_name\_\_ will be set to the module’s name.  
**\_\_name\_\_ is a built-in variable which evaluates to the name of the current module.** Thus it can be used to check whether the current script is being run on its own or being imported somewhere else by combining it with if statement, as shown below.

Consider two separate files File1 and File2.

|  |
| --- |
| # File1.py  print "File1 \_\_name\_\_ = %s" %\_\_name\_\_    if \_\_name\_\_ == "\_\_main\_\_":      print "File1 is being run directly"  else:      print "File1 is being imported" |

# File2.py

import File1

print "File2 \_\_name\_\_ = %s" %\_\_name\_\_

if \_\_name\_\_ == "\_\_main\_\_":

    print "File1 is being run directly"

else:

    print "File1 is being imported"

Now the interpreter is given the command to run File1.py.

**python File1.py**

**Output :**

File1 \_\_name\_\_ = \_\_main\_\_

File1 is being run directly

And then File2.py is run.

**python File2.py**

**Output :**

File1 \_\_name\_\_ = File1

File1 is being imported

File2 \_\_name\_\_ = \_\_main\_\_

File2 is being run directly

As seen above, when File1.py is run directly, the interpreter sets the \_\_name\_\_ variable as \_\_main\_\_ and when it is run through File2.py by importing, the \_\_name\_\_ variable is set as the name of the python script, i.e. File1. Thus, it can be said that **if \_\_name\_\_ == “\_\_main\_\_” is the part of the program that runs when the script is run from the command line using a command like python File1.py.**

**Class variables:**

**These are the variables which are common for all instances or objects of a class.**

**Example:**

**Class employes:**

**Count=0**

**Def \_\_init\_\_(self, name, salary):**

**Self.name=name**

**Self.salary=salary**

**Employee.count=employee.count+1**

**Def displaycount(self):**

**Print(“total employees”,employee.count)**

**Def display(self):**

**Print(self.name, self.salary)**

**o/p:**

**The count variable is a class variable whose value is shared among all objects of the class. It can be accessed using class name or using object either inside or outside the class.**

**Emp1=employee(“Zam”,2000)**

**Emp2=employee(“mani”,5000)**

**Emp1.display()**

**Emp2.display()**

**Print(“total employees “,employee.count)**

**o/p:**

**destructor:**

**A class can implement a special method \_\_del\_\_() called a destructor. It is invoked when the instance or object of a class is to be destroyed.**

**Example1:**

**class point:**

**def \_\_init\_\_(self, x=1, y=2):**

**self.x=x**

**self.y=y**

**print("created")**

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

**print(point, "destroyed")**

**p1=point()**

**p2=p1**

**p3=p1**

**print(p1.x,p1.y)**

**del p1**

**print(p2.x,p2.y)**

**del p2**

**print(p3.x,p3.y)**

**del p3**

**o/p:**

created

1 2

1 2

1 2

<class '\_\_main\_\_.point'> destroyed

**The destroyer will be called after the program ended. When all the references to object are deleted.**

**The \_\_del\_\_ () method is the destructor method which is used for garbage collection for handling memory management automatically. It is called automatically when all the references to the object has been deleted.**

**By using del keyword we delete the references, there by destructor is invoked automatically.**

**Example2:**

**class emp:**

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

**print("employee initialized: constructor called")**

**def display(self):**

**print("display method executing")**

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

**print("destructor called")**

**def create\_obj():**

**print("making object")**

**inobj=emp()**

**print("function ends")**

**return inobj**

**print("calling create\_obj function")**

**outobj=create\_obj()**

**outobj.display()**

**del outobj**

**print("end")**

**o/p:**

calling create\_obj function

making object

employee initialized: constructor called

function ends

display method executing

destructor called

end

**python inheritance:**

**It enables us to define a class that takes all the functionality from parent class. Here the parent class is also called base class and the new class is called derived class.**

**Syntax:**

**Class baseclass:**

**Body of the baseclass**

**Class derivedclass(baseclass):**

**Body of the derivedclass**

**Types of inheritance:**

1. **single inheritance**
2. **multiple inheritance**
3. **multilevel inheritance**

**single inheritance:**

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.

Python Inheritance

**Syntax**

class derived-class(base class):

<class-suite>

**example:  
class base:**

**def \_\_init\_\_(self, fname, lname):**

**self.firstname = fname**

**self.lastname = lname**

**def printname(self):**

**print(self.firstname, self.lastname)**

**#Use the base class to create an object, and then execute the #printname method:**

**x = Person("John", "Doe")**

**x.printname()**

**class Student(Person):**

**def disp(self):**

**print(“derived class function”)**

**x = Student("Mike", "Olsen")**

**x.printname()**

**x.disp()**

**o/p:**

John Doe  
Mike Olsen

**Derived class function**

Example2:

class Animal:

    def speak(self):

        print("Animal Speaking")

#child class Dog inherits the base class Animal

class Dog(Animal):

    def bark(self):

        print("dog barking")

d = Dog()

d.bark()

d.speak()

o/p:

dog barking

Animal Speaking

## Python Multi-Level inheritance

Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

Python Inheritance

**Syntax**

class bclass1:

    <class-suite>

class dclass2(bclass1):

    <class suite>

class dclass3(dclass2):

    <class suite>

Example1:

class Animal:

def speak(self):

print("Animal Speaking")

#The child class Dog inherits the base class Animal

class Dog(Animal):

def bark(self):

print("dog barking")

#The child class Dogchild inherits another child class Dog

class DogChild(Dog):

def eat(self):

print("Eating bread...")

d = DogChild()

d.bark()

d.speak()

d.eat()

o/p:

dog barking

Animal Speaking

Eating bread...

## Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.

Python Inheritance

Syntax

class Base1:

<class-suite>

class Base2:

<class-suite>

.

.

.

class BaseN:

<class-suite>

class Derived(Base1, Base2, ...... BaseN):

<class-suite>

**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(d.Summation(10,20))

print(d.Multiplication(10,20))

print(d.Divide(10,20))

**Output:**

30

200

0.5

**The issubclass(sub,sup) method**

The issubclass(sub, sup) method 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 otherwise returns false.

Consider the following example.

**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

**The isinstance (obj, class) method**

The isinstance() method 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.**

Consider the following example.

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))

Output:

True

**Method Overriding**

**When the method name in base class and in derived class are same.**

**Then while creating object for derived class, by default the method of derived class will override the method of base class.**

When the 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.

Consider the following example to perform method overriding in python.

**Example**

class Animal:

def speak(self):

print("speaking")

class Dog(Animal):

def speak(self):

print("Barking")

d = Dog()

d.speak()

b=Animal()

b.speak()

Animal.speak()

Dog.speak()

Output:

Barking

Speaking

Speaking

Barking

**Real Life Example of method overriding**

class Bank:

def getroi(self):

return 10;

class SBI(Bank):

def getroi(self):

return 7;

class ICICI(Bank):

def getroi(self):

return 8;

b1 = Bank()

b2 = SBI()

b3 = ICICI()

print("Bank Rate of interest:",b1.getroi());

print("SBI Rate of interest:",b2.getroi());

print("ICICI Rate of interest:",b3.getroi());

Output:

Bank Rate of interest: 10

SBI Rate of interest: 7

ICICI Rate of interest: 8

**Method Resolution Order (MRO) in Python**

Method Resolution Order (MRO) is the order in which Python looks for a method in a hierarchy of classes. Especially it plays vital role in the context of multiple inheritance as single method may be found in multiple super classes.

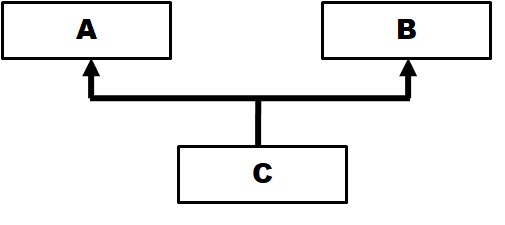
To understand the concept of MRO and its need, lets examine a few cases.

## Case 1

This is a simple case where we have class C derived from both A and B. When method process() is called with object of class C then process() method in class A is called.

Python constructs the order in which it will look for a method in the hierarchy of classes. It uses this order, known as MRO, to determine which method it actually calls.

It is possible to see MRO of a class using **mro()** method of the class.



class A:

def process(self):

print('A process()')

class B:

pass

class C(A, B):

pass

obj = C()

obj.process()

print(C.mro()) # print MRO for class C

The above diagram illustrates hierarchy of classes.

When run, the above program displays the following output:

A process()

[<class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.B'>, <class 'object'>]

From MRO of class C, we get to know that Python looks for a method first in class C. Then it goes to A and then to B. So, first it goes to super class given first in the list then second super class, from left to right order. Then finally Object class, which is a super class for all classes.

## Case 2

Now, let’s make it a little more complicated by adding process() method to class B also.

class A:

def process(self):

print('A process()')

class B:

def process(self):

print('B process()')

class C(A, B):

pass

obj = C()

obj.process()

When you run the above code, it prints the following:

A process()

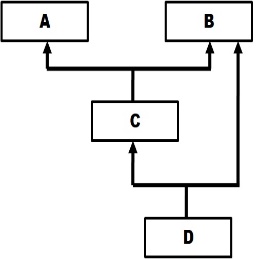
Python calls process() method in class A. According to MRO, it searches A first and then B. So if method is found in A then it calls that method.

However, if we remove process() method from class A then process() method in class B will be called as it is the next class to be searched according to MRO.

**\*The ambiguity that arises from multiple inheritance is handled by Python using MRO.**

## Case 3

In this case, we create D from C and B. Classes C and B have process() method and as expected MRO chooses method from C. Remember it goes from left to right. So it searches C first and all its super classes of C and then B and all its super classes. We can observe that in MRO of the output given below.



class A:

def process(self):

print('A process()')

class B:

def process(self):

print('B process()')

class C(A, B):

def process(self):

print('C process()')

class D(C,B):

pass

obj = D()

obj.process()

print(D.mro())

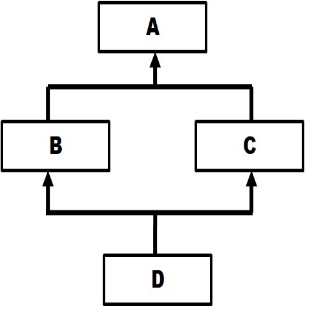
Running the above program will produce the following output:

C process()

[<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.B'>, <class 'object'>]

## Case 4

Now, lets change the hierarchy. We create B and C from A and then D from B and C. Method process() is present in both A and C.



class A:

def process(self):

print('A process()')

class B(A):

pass

class C(A):

def process(self):

print('C process()')

class D(B,C):

pass

obj = D()

obj.process()

Output of the above program is:

C process()

[<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]

Ex2:

class A:

def process(self):

print('A process()')

class B(A):

pass

class C(A):

pass

class D(B,C):

pass

obj = D()

obj.process()

print(D.mro())

o/p:

A process()

[<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]

When we call process() with an object of class D, it should start with first Super class – B (and its super classes) and then second super class – C (and its super classes). If that is the case then we will call process() method from class A as B doesn’t have it and A is super class for B.

However, that is contradictory to rule of inheritance, as most specific version must be taken first and then least specific (generic) version. So, calling process() from A, which is super class of C, is not correct as C is a direct super class of D. That means C is more specific than A. So method must come from C and not from A.

This is where Python applies a simple rule that says (known as good head question) **when in MRO we have a super class before subclass then it must be removed from that position in MRO.**

So the original MRO will be:

D -> B -> A -> C -> A

If you include object class also in MRO then it will be:

D -> B-> A -> object -> C -> A -> object

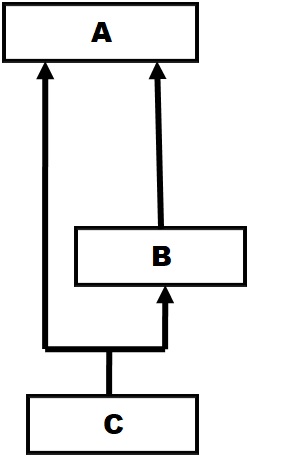
But as A is super class of C, it cannot be before C in MRO. So, Python removes A from that position, which results in new MRO as follows:

D -> B -> C -> A -> object

The output of the above program proves that.

## Case 5

There are cases when Python cannot construct MRO owing to complexity of hierarchy. In such cases it will throw an error as demonstrated by the following code.



class A:

def process(self):

print('A process()')

class B(A):

def process(self):

print('B process()')

class C(A, B):

pass

obj = C()

obj.process()

When you run the above code, the following error is shown:

TypeError: Cannot create a consistent method resolution

order (MRO) for bases A, B

The problem comes from the fact that class A is a super class for both C and B. If you construct MRO then it should be like this:

C -> A -> B -> A

Then according to the rule (good head) A should NOT be ahead of B as A is super class of B. So new MRO must be like this:

C -> B -> A

But A is also direct super class of C. So, if a method is in both A and B classes then which version should class C call? According to new MRO, the version in B is called first ahead of A and that is not according to inheritance rules (specific to generic) resulting in Python to throw error.

Understanding MRO is very important for any Python programmer. I strongly recommend trying more cases until you completely understand how Python constructs MRO. Do not confuse yourself by taking old way of constructing MRO used in earlier versions of Python. It is better to consider only Python 3.

**METHOD/FUNCTION OVERLOADING:**

Python does not support method overloading. We may overload the methods but can only use the latest defined method.

Ex1:

class A:

def fun1():

print(“sample message1”)

def fun1(a):

print(“the data is”,a)

def fun1(a,b):

print(“the data are”,a,b)

x=A()

x.fun1() //error

x.fun1(10) //error

x.fun1(10,20) //executed

x.fun1(‘A’,’B’) //executed

o/p:

the data are 10 20

the data are A B

ex:

|  |
| --- |
| #!/usr/bin/env python  class Human:   def sayHello(self, name=None):    if name is not None:  print('Hello ' + name)  else:  print('Hello ')   # Create instance obj = Human()   # Call the method obj.sayHello()   # Call the method with a parameter obj.sayHello('Guido') |

Output:

|  |
| --- |
| Hello Hello Guido |

ex2;

|  |
| --- |
| # First product method.  # Takes two argument and print their  # product  def product(a, b):      p = a \* b      print(p)    # Second product method  # Takes three argument and print their  # product  def product(a, b, c):      p = a \* b\*c      print(p)    # Uncommenting the below line shows an error  # product(4, 5)    # This line will call the second product method  product(4, 5, 5) |

Output:

100

Ex3:

However we may use other implementation in python to make the same function work differently i.e. as per the arguments.

|  |
| --- |
| Program:  # Function to take multiple arguments  def add(datatype, \*args):        # if datatype is int      # initialize answer as 0      if datatype =='int':          answer = 0        # if datatype is str      # initialize answer as ''      if datatype =='str':          answer =''        # Traverse through the arguments      for x in args:            # This will do addition if the          # arguments are int. Or concatenation          # if the arguments are str          answer = answer + x        print(answer)    # Integer  add('int', 5, 6)    # String  add('str', 'Hi ', 'Alok') |

Output:

11

Hi Alok

**Data abstraction in python**

Abstraction is an important aspect of object-oriented programming. In python, we can also perform data hiding by adding the double underscore (\_\_\_) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.

Consider the following example.

Example1:

class Employee:

\_\_count = 0;

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

Employee.\_\_count = Employee.\_\_count+1

def display(self):

print("The number of employees",Employee.\_\_count)

emp = Employee()

emp2 = Employee()

try:

print(emp.\_\_count)

finally:

emp.display()

Output:

The number of employees 2

AttributeError: 'Employee' object has no attribute '\_\_count'

**OPERATOR OVERLOADING:**

### What is operator overloading

Operator overloading as evident from the name itself means the ability to overload the operator to provide extra functionality in addition to its real operational meaning. For example ‘+’ operator which is used with numbers to perform addition operation. But ‘+’ operator when used with two strings concatenate those Strings and merge two lists when used with lists in Python. It is possible because ‘+’ operator is overloaded in str and list class to provide extended functionality.

### Python Operator overloading for ‘+’ operator

#using + operator with integers to add them

print(5 + 7)

#using + operator with Strings to concatenate them

print('hello ' + 'world')

a = [1, 2, 3]

b = [4, 5, 6]

# using + operator with List to concatenate them

print(a + b)

**Output**

12

hello world

[1, 2, 3, 4, 5, 6]

### When is Operator overloading required?

if you want to use operator with custom objects. For example if you want to use ‘+’ operator with your custom class objects.

In the example there is a class Point with two variables x and y. Two objects of Point class are instantiated and you try to add those objects with the intention to add the data (p1.x + p2.x) and (p1.y + p2.y) of these two objects.

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

p1 = Point(1, 2)

p2 = Point(3, 4)

print(p1+p2)

**Output**

Traceback (most recent call last):

File "F:/NETJS/NetJS\_2017/Python/Test/Test.py", line 9, in <module>

print(p1+p2)

TypeError: unsupported operand type(s) for +: 'Point' and 'Point'

For all operators internally Python defines methods to provide functionality for those operators. For example functionality for ‘+’ operator is provide by special method **\_\_add\_\_()**. Whenever ‘+’ operator is used internally \_\_add\_\_() method is invoked to do the operation.

Internal methods that provide functionality for the operators are known as **magic methods in Python**. These magic methods are automatically invoked when corresponding operators are used.

**Overloading ‘+’ operator to work with custom objects**

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_add\_\_(self, other):

return self.x + other.x, self.y + other.y

p1 = Point(1, 2)

p2 = Point(3, 4)

print(p1+p2)

**Output**

(4, 6)

What actually happens is that, when you do p1 + p2, Python will call p1.\_\_add\_\_(p2) which in turn is Point.\_\_add\_\_(p1,p2). Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

|  |  |  |
| --- | --- | --- |
| Operator Overloading Special Functions in Python | | |
| **Operator** | **Expression** | **Internally** |
| Addition | p1 + p2 | p1.\_\_add\_\_(p2) |
| Subtraction | p1 - p2 | p1.\_\_sub\_\_(p2) |
| Multiplication | p1 \* p2 | p1.\_\_mul\_\_(p2) |
| Power | p1 \*\* p2 | p1.\_\_pow\_\_(p2) |
| Division | p1 / p2 | p1.\_\_truediv\_\_(p2) |
| Floor Division | p1 // p2 | p1.\_\_floordiv\_\_(p2) |
| Remainder (modulo) | p1 % p2 | p1.\_\_mod\_\_(p2) |
| Bitwise Left Shift | p1 << p2 | p1.\_\_lshift\_\_(p2) |
| Bitwise Right Shift | p1 >> p2 | p1.\_\_rshift\_\_(p2) |
| Bitwise AND | p1 & p2 | p1.\_\_and\_\_(p2) |
| Bitwise OR | p1 | p2 | p1.\_\_or\_\_(p2) |
| Bitwise XOR | p1 ^ p2 | p1.\_\_xor\_\_(p2) |
| Bitwise NOT | ~p1 | p1.\_\_invert\_\_() |
| Matrix Multiplication. | @ | \_\_matmul\_\_(self, other) |

|  |  |  |
| --- | --- | --- |
| Comparison Operator Overloading in Python | | |
| **Operator** | **Expression** | **Internally** |
| Less than | p1 < p2 | p1.\_\_lt\_\_(p2) |
| Less than or equal to | p1 <= p2 | p1.\_\_le\_\_(p2) |
| Equal to | p1 == p2 | p1.\_\_eq\_\_(p2) |
| Not equal to | p1 != p2 | p1.\_\_ne\_\_(p2) |
| Greater than | p1 > p2 | p1.\_\_gt\_\_(p2) |
| Greater than or equal to | p1 >= p2 | p1.\_\_ge\_\_(p2) |

**Magic methods for compound assignment operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Magic Method** | **Description** |
| += | \_\_iadd\_\_(self, other) | Addition assignment |
| –= | \_\_isub\_\_(self, other) | Subtraction assignment |
| \*= | \_\_imul\_\_(self, other) | Multiplication assignment |
| /= | \_\_itruediv\_\_(self, other) | Division assignment |
| %= | \_\_imod\_\_(self, other) | Modulus assignment |
| //= | \_\_ifloordiv\_\_(self, other) | Division with integer result, discarding any fractional part |
| \*\*= | \_\_ipow\_\_(self, other) | Return a to the power b, for a and b numbers. |
| @= | \_\_imatmul\_\_(self, other) | Matrix Multiplication. |

### Overloading ‘\*’ operator in Python

class Point:

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

self.x = x

#overriding magic method

def \_\_mul\_\_(self, other):

return self.x \* other.x

p1 = Point(12)

p2 = Point(5)

print(p1\*p2)

**Output**

60

### Overloading comparison operator (>) in Python

class Person:

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

self.name = name

self.salary = salary

#overriding magic method

def \_\_gt\_\_(self, other):

return self.salary > other.salary

obj1 = Person('John', 4500)

obj2 = Person('Natasha', 6000)

print(obj1.name, 'earns more than', obj2.name, '-', obj1 > obj2)

**Output**

John earns more than Natasha – False

**MATRIX PROGRAMS:**

**Add Two Matrices**

X = [[1,2,3], [4,5,6], [7,8,9]]

Y = [[10,11,12],

[13,14,15],

[16,17,18]]

Result = [[0,0,0],

[0,0,0],

[0,0,0]]

# iterate through rows

for i in range(len(X)):

# iterate through columns

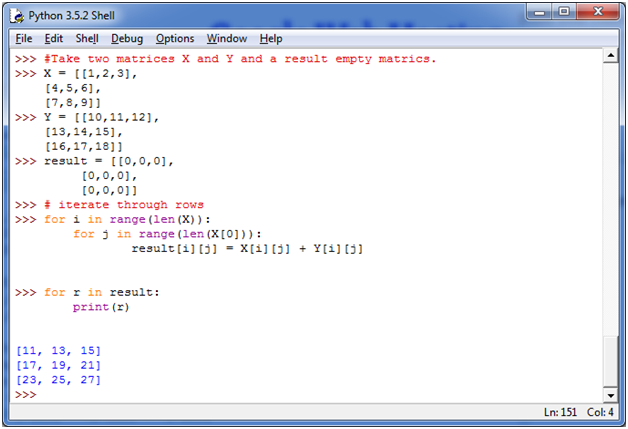
for j in range(len(X[0])):

result[i][j] = X[i][j] + Y[i][j]

for r in result:

print(r)

O/P:



Multiply Two Matrices

X = [[1,2,3],

[4,5,6],

[7,8,9]]

Y = [[10,11,12],

[13,14,15],

[16,17,18]]

Result = [[0,0,0],

[0,0,0],

[0,0,0]]

# iterate through rows of X

for i in range(len(X)):

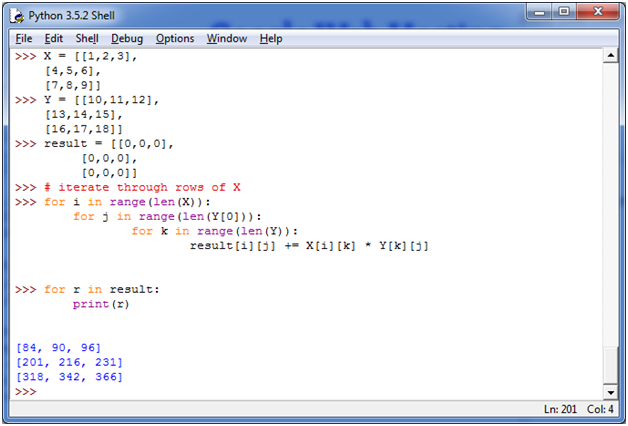
for j in range(len(Y[0])):

for k in range(len(Y)):

result[i][j] += X[i][k] \* Y[k][j]

for r in result:

print(r)



**Transpose a Matrix**

X = [[1,2],

[4,5],

[7,8]]

Result = [[0,0,0],

[0,0,0]]

# iterate through rows

for i in range(len(X)):

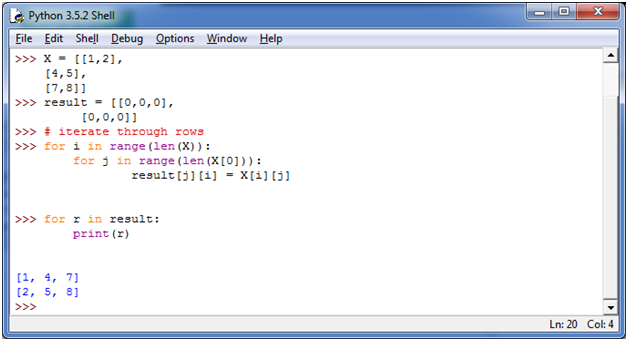
for j in range(len(X[0])):

result[j][i] = X[i][j]

for r in result:

print(r)

Output:



**Sort Words in Alphabetic Order**

my\_str = input("Enter a string: ")

# breakdown the string into a list of words

words = my\_str.split()

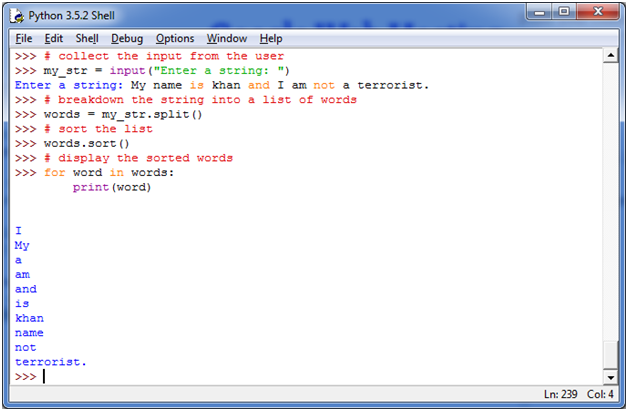
# sort the list

words.sort()

# display the sorted words

for word in words:

print(word)



**Remove Punctuation from a String**

# define punctuation

punctuation = '''''!()-[]{};:'"\,<>./?@#$%^&\*\_~'''

# take input from the user

my\_str = input("Enter a string: ")

# remove punctuation from the string

no\_punct = ""

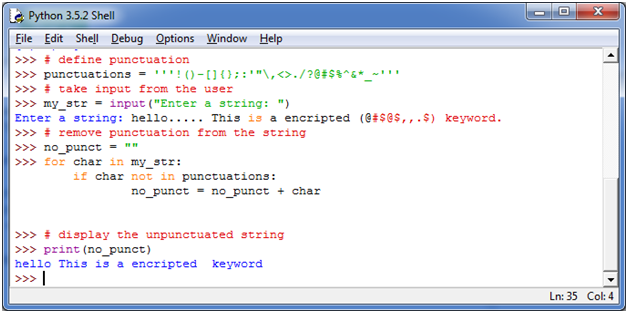
for char in my\_str:

if char not in punctuation:

no\_punct = no\_punct + char

# display the unpunctuated string

print(no\_punct)



# **NumPy Arrays**

### NumPy is a package for scientific computing which has support for a powerful N-dimensional array object.

NumPy provides multidimensional array of numbers (which is actually an object). Let's take an example:

import numpy as np

a = np.array([1, 2, 3])

print(a) # Output: [1, 2, 3]

print(type(a)) # Output: <class 'numpy.ndarray'>

As you can see, NumPy's array class is called ndarray.

**1. Array of integers, floats and complex Numbers**

import numpy as np

A = np.array([[1, 2, 3], [3, 4, 5]])

print(A)

A = np.array([[1.1, 2, 3], [3, 4, 5]]) # Array of floats

print(A)

A = np.array([[1, 2, 3], [3, 4, 5]], dtype = complex) # Array of complex numbers

print(A)

When you run the program, the output will be:

[[1 2 3]

[3 4 5]]

[[1.1 2. 3. ]

[3. 4. 5. ]]

[[1.+0.j 2.+0.j 3.+0.j]

[3.+0.j 4.+0.j 5.+0.j]]

**2. Array of zeros and ones**

import numpy as np

zeors\_array = np.zeros( (2, 3) )

print(zeors\_array)

'''

Output:

[[0. 0. 0.]

[0. 0. 0.]]

'''

ones\_array = np.ones( (1, 5), dtype=np.int32 ) # specifying dtype

print(ones\_array)

# Output: [[1 1 1 1 1]]

### Here, we have specified dtype to 32 bits (4 bytes).

### Hence, this array can take values from -2-31 to 2-31-1.

### arange():

### It will create arrays with regularly incrementing values.

np.arange() can generate a sequence of numbers given the start and end.

## Generated a NumPy array from 0 to 4

print(np.arange(0,5))

Ex1:

import numpy as np

A = np.arange(4)

print('A =', A)

B = np.arange(12)

print('B =', B)

o/p:

A = [0 1 2 3]

B = [ 0 1 2 3 4 5 6 7 8 9 10 11]

Ex2:

import numpy as np

A = np.arange(4)

print('A =', A)

B = np.arange(12).reshape(2,6)

print('B =', B)

o/p:

A = [0 1 2 3]

B = [[ 0 1 2 3 4 5]

[ 6 7 8 9 10 11]]

Ex3:

import numpy as np

A=np.arange(10)

print(A)

B=np.arange(2, 10, dtype=float)

print(B)

C=np.arange(2, 3, 0.1)

print(C)

o/p:

[0 1 2 3 4 5 6 7 8 9]

[2. 3. 4. 5. 6. 7. 8. 9.]

[2. 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9]

## Shape of NumPy array

## A shape is a tuple of the format (n\_rows, n\_cols)

Ex1:

x=np.array([np.arange(0,5), np.arange(5,10)])

print(x)

print(x.shape)

o/p:

[[0 1 2 3 4]

[5 6 7 8 9]]

(2, 5)

Ex2:

x=np.array([np.arange(0,5,.5), np.arange(5,10)])

print(x)

print(x.shape)

o/p:

[array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])

array([5, 6, 7, 8, 9])]

(2,)

**Addition of Two Matrices**

We use + operator to add corresponding elements of two NumPy matrices.

import numpy as np

A = np.array([[2, 4], [5, -6]])

B = np.array([[9, -3], [3, 6]])

C = A + B # element wise addition

print(C)

'''

Output:

[[11 1]

[ 8 0]]

'''

### Subtraction

Subtraction is similar to addition. We just need to change the operation from addition to subtraction.

Ex:

import numpy as np

## Generate two matrices

A= np.array([np.arange(0,3), np.arange(3,6)])

B = np.array([np.arange(6,9), np.arange(9,12)])

print("Matrix1: n ", A)

print("Matrix2: n ", B)

## Subtract 1 from each element in mat\_2d\_1 and print it

print("Scalar subtraction: n ", A - 1)

## Subtract two matrices above elementwise

print("Element wise subtraction of two matrices of same size: n ")

print(A- B)

o/p:

Matrix1: n

[[0 1 2]

[3 4 5]]

Matrix2: n

[[ 6 7 8]

[ 9 10 11]]

Scalar subtraction: n

[[-1 0 1]

[ 2 3 4]]

Element wise subtraction of two matrices of same size: n

[[-6 -6 -6]

[-6 -6 -6]]

**Product**

Two types of multiplication or product operation can be done on NumPy matrices

* Scalar product: A scalar value is multiplied with all elements of a matrix
* Dot product: This is the product of two matrices as per the rules of matrix multiplication.

Ex1:

import numpy as np

A= np.array([np.arange(0,3), np.arange(3,6)])

B = np.array([np.arange(0,2), np.arange(2,4), np.arange(4,6)])

## Print shapes and matrices

print("Matrix1: n ", A)

print("Matrix1 shape: n", A.shape)

print("Matrix2: n ",B)

print("Matrix2 shape: n", B.shape)

## Multiply each element by 2 in mat\_2d\_1 and print it

print("Scalar Product: n ")

print(A \* 2)

## Find product of two matrices above using dot product

print("Dot Product: n ")

print(np.dot(A, B))

o/p:

Matrix1: n

[[0 1 2]

[3 4 5]]

Matrix1 shape: n (2, 3)

Matrix2: n

[[0 1]

[2 3]

[4 5]]

Matrix2 shape: n (3, 2)

Scalar Product: n

[[ 0 2 4]

[ 6 8 10]]

Dot Product: n

[[10 13]

[28 40]]

**Transpose of a Matrix**

We use [numpy.transpose](https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.transpose.html) to compute transpose of a matrix.

import numpy as np

A = np.array([[1, 1], [2, 1], [3, -3]])

print(A.transpose())

'''

Output:

[[ 1 2 3]

[ 1 1 -3]]

'''

### Python Exception

**Types of errors**

Errors in your Python programs can be categorized into

at least two types-

* Syntax errors
* Exceptions

Syntax errors or compile time errors are the errors due to which your program fails to compile. Such errors are detected at the compile time itself, file name, line number and error description is displayed so you know where to look for the error and how to correct it.

For example if you don’t write colon after if statement-

def check\_equality(str1, str2):

if str1 > str2

print(str1, 'is greater than', str2)

### Exceptions

Even if a statement or expression is syntactically correct it may result in an error at run time. Errors detected during runtime are called exceptions and they disrupt the normal execution flow of the program.

For example in the following program there is an attempt to divide by zero which results in ZeroDivisionError at runtime.

def divide\_num(num1, num2):

return num1/num2

divide\_num(10, 0)

An exception can be defined as an abnormal condition in a program resulting in the disruption in the flow of the program.

Exceptions are 5 types:

1. Zero Division Exception
2. Name Error
3. Indentation Error
4. IO Error
5. EOF Error

**Exception Handling in Python using Try Block:**

Here the code which might have exception have to place in try block. The keywords used are **try, except, finally**

Syntax:

try:

suspicious code

except exception1:

block1 code

except exception2:

block2 code

Finally:

Other code

Ex1: example of ZeroDivisionError

try:

x = int(input('Enter the first number: '))

y = int(input('Enter the second number: '))

print(x/y)

except ZeroDivisionError:

print ('The second number cannot be zero!')

o/p:

Enter the first number: 10

Enter the second number: 0

The second number cannot be zero!

Ex2:example of ZeroDivisionError

try:

x = int(input('Enter the first number: '))

y = int(input('Enter the second number: '))

z=(x/y)

print("a/b = %d"%z)

except ZeroDivisionError:

print("cannot divide by zero")

else:

print("division successful")

print("end of program")

o/p:

Enter the first number: 10

Enter the second number: 2

a/b = 5

division successful

end of program

ex3: Example of IOError

try: #try block

fileptr=open("file7.txt","r")

except IOError: #except block

print("the file is not found")

else: #else block

print("the file opened successfully")

fileptr.close()

print("end of program") #other code block

o/p:

the file opened successfully

end of program

**Finally:**

A final block will be executed regardless of an error. It is an optional clause. It executes generally for external resources. It will execute even if exception exist or else not exist.

Ex1: Example of IOError

try:

fileptr=open("file1.txt","r")

except IOError:

print("the file is not found")

else:

print("the file opened successfully")

fileptr.close()

finally:

print("i am finally")

print("end of program")

o/p:

the file is not found

i am finally

end of program

ex2: Example of IOError

try:

fileptr=open("file7.txt","r")

#fileptr.write("sample text")

f=fileptr.read()

print(f)

except IOError:

print("the file error")

else:

print("let us read")

finally:

fileptr.close()

print("file closed")

o/p:

this is file seven

let us read

file closed

ex3: Example of IOError

try:

fileptr=open("file7.txt","r")

fileptr.write("sample text")

except IOError:

print("the file error")

else:

print("let us read")

finally:

fileptr.close()

print("file closed")

o/p:

the file error

file closed

ex4: Example of ValueError

def inputnumber():

try:

r=int(input("enter a no"))

except ValueError:

print("wrong entry")

else:

print("right entry")

finally:

print("end of try")

inputnumber()

o/p:

enter a noAA

wrong entry

end of try

ex5: Example of ValueError

def inputnumber():

while(True):

try:

r=int(input("enter a no"))

except ValueError:

print("wrong entry")

continue

else:

print("right entry")

break

finally:

print("end of exceptions")

inputnumber()

o/p:

enter a noA

wrong entry

end of exceptions

enter a noA

wrong entry

end of exceptions

enter a no2

right entry

end of exceptions

**raise:**

An exception can be raised using raise clause in python.

Syntax:

Raise exception\_class <value>

Ex1: example of ValueError

try:

age=int(input("enter age"))

if(age<18):

raise ValueError;

else:

print("the age is valid")

except ValueError:

print("the age is invalid")

finally:

print("end of exceptions")

o/p:

enter age14

the age is invalid

ex2: example of ZeroDivisionError

try:

a=int(input("enter a"))

b=int(input("enter b"))

if(b==0):

raise ZeroDivisionError;

else:

print("a/b =",a/b)

except ZeroDivisionError:

print("b cannot be 0")

o/p:

enter a12

enter b0

b cannot be 0

**Some important built in exceptions are:**

AttributeError, EOFError, FloatingPointError, ImportError, IndexError, KeyError,MemoryError, NameError, OverlfowError, RuntimeError, SyntaxError, IndentationError, TypeError, ValueError, ZeroDivisionError

**Python Custom Exceptions/**

**User Defined Exception:**

In python we can create custom exceptions by creating a new class. This exception class ahs to be derived either directly or indirectly.

Ex1:

class error(Exception):

pass

class Derived\_smallvalue(error):

pass

class Derived\_largevalue(error):

pass

n=10

while True:

try:

i=int(input("enter no"))

if(i<n):

raise Derived\_smallvalue

elif(i>n):

raise Derived\_largevalue

break

except Derived\_smallvalue:

print("it is less than 10")

except Derived\_largevalue:

print("it is greater than 10")

print("congrats, it is", 10)

o/p:

enter no 11

It is greater than 10

**Python RegEx**

A **Re**gular **Ex**pression (RegEx) is a sequence of characters that defines a search pattern. For example,

^a...s$

The above code defines a RegEx pattern. The pattern is: **any five letter string starting with *a* and ending with *s***.

A pattern defined using RegEx can be used to match against a string.

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| ^a...s$ | abs | No match |
| alias | Match |
| abyss | Match |
| Alias | No match |
| An abacus | No match |

Python has a module named re to work with RegEx. Here's an example:

Ex:

import re

pattern = '^a...s$'

test\_string = 'abhas'

result = re.match(pattern, test\_string)

if result:

print("Search successful.")

else:

print("Search unsuccessful.")

o/p:

Search Successful

Here, we used re.match() function to search *pattern* within the *test\_string*. The method returns a match object if the search is successful. If not, it returns None.

### Meta Characters

Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters:

**[]** **.** **^** **$** **\*** **+** **?** **{}** **()** **\** **|**

**[] - Square brackets**

Square brackets specifies a set of characters you wish to match.

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| [abc] | a | 1 match |
| ac | 2 matches |
| Hey Jude | No match |
| abc de ca | 5 matches |

Here, [abc] will match if the string you are trying to match contains any of the a, b or c.

You can also specify a range of characters using - inside square brackets.

* [a-e] is the same as [abcde].
* [1-4] is the same as [1234].
* [0-39] is the same as [01239].

You can complement (invert) the character set by using caret ^ symbol at the start of a square-bracket.

* [^abc] means any character except *a* or *b* or *c*.
* [^0-9] means any non-digit character.
* . - **Period**
* A period matches any single character (except newline '\n').

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| .. | a | No match |
| ac | 1 match |
| acd | 1 match |
| acde | 2 matches (contains 4 characters) |

^ - **Caret**

The caret symbol ^ is used to check if a string **starts with** a certain character.

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| ^a | a | 1 match |
| abc | 1 match |
| bac | No match |
| ^ab | abc | 1 match |
| acb | No match (starts with a but not followed by b) |

$ - **Dollar**

The dollar symbol $ is used to check if a string **ends with** a certain character.

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| a$ | a | 1 match |
| formula | 1 match |
| cab | No match |

\* - **Star**

The star symbol \* matches **zero or more occurrences** of the pattern left to it.

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| ma\*n | mn | 1 match |
| man | 1 match |
| maaan | 1 match |
| main | No match (a is not followed by n) |
| woman | 1 match |

+ - **Plus**

The plus symbol + matches **one or more occurrences** of the pattern left to it.

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| ma+n | mn | No match (no a character) |
| man | 1 match |
| maaan | 1 match |
| main | No match (a is not followed by n) |
| woman | 1 match |

? - **Question Mark**

The question mark symbol ? matches **zero or one occurrence** of the pattern left to it.

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| ma?n | mn | 1 match |
| man | 1 match |
| maaan | No match (more than one a character) |
| main | No match (a is not followed by n) |
| woman | 1 match |

{} - **Braces**

Consider this code: {n,m}. This means at least *n*, and at most *m* repetitions of the pattern left to it.

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| a{2,3} | abc dat | No match |
| abc daat | 1 match (at daat) |
| aabc daaat | 2 matches (at aabc and daaat) |
| aabc daaaat | 2 matches (at aabc and daaaat) |

* Let's try one more example. This RegEx [0-9]{2, 4} matches at least 2 digits but not more than 4 digits

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| [0-9]{2,4} | ab123csde | 1 match (match at ab123csde) |
| 12 and 345673 | 2 matches (at 12 and 345673) |
| 1 and 2 | No match |

| - **Alternation**

Vertical bar | is used for alternation (or operator).

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| a|b | cde | No match |
| ade | 1 match (match at ade) |
| acdbea | 3 matches (at acdbea) |

* Here, a|b match any string that contains either *a* or *b*

() - **Group**

Parentheses () is used to group sub-patterns. For example, (a|b|c)xz match any string that matches either *a* or *b* or *c* followed by *xz*

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| (a|b|c)xz | ab xz | No match |
| abxz | 1 match (match at abxz) |
| axz cabxz | 2 matches (at axzbc cabxz) |

**Special Sequences:**

\A - Matches if the specified characters are at the start of a string.

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| \Athe | the sun | Match |
| In the sun | No match |

\b - Matches if the specified characters are at the beginning or end of a word.

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| \bfoo | football | Match |
| a football | Match |
| afootball | No match |
| foo\b | the foo | Match |
| the afoo test | Match |
| the afootest | No match |

\B - Opposite of \b. Matches if the specified characters are **not** at the beginning or end of a word.

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| \Bfoo | football | No match |
| a football | No match |
| afootball | Match |
| foo\B | the foo | No match |
| the afoo test | No match |
|  | the afootest | Match |
|  |  |  |

\s - Matches where a string contains any whitespace character. Equivalent to [ \t\n\r\f\v].

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| \s | Python RegEx | 1 match |
| PythonRegEx | No match |

\d - Matches any decimal digit. Equivalent to [0-9]

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| \d | 12abc3 | 3 matches (at 12abc3) |
| Python | No match |

\D - Matches any non-decimal digit. Equivalent to [^0-9]

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| \D | 1ab34"50 | 3 matches (at 1ab34"50) |
| 1345 | No match |

w - Matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9\_]. By the way, underscore \_ is also considered an alphanumeric character.

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| \w | 12&": ;c | 3 matches (at 12&": ;c) |
| %"> ! | No match |

\W - Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9\_]

| **Expression** | **String** | **Matched?** |
| --- | --- | --- |
| \W | 1a2%c | 1 match (at 1a2%c) |
| Python | No match |

## RegEx Functions

The re module offers a set of functions that allows us to search a string for a match:

|  |  |
| --- | --- |
| **Function** | **Description** |
| [findall](https://www.w3schools.com/python/python_regex.asp#findall) | Returns a list containing all matches |
| [search](https://www.w3schools.com/python/python_regex.asp#search) | Returns a [Match object](https://www.w3schools.com/python/python_regex.asp#matchobject) if there is a match anywhere in the string |
| [split](https://www.w3schools.com/python/python_regex.asp#split) | Returns a list where the string has been split at each match |
| [sub](https://www.w3schools.com/python/python_regex.asp#sub) | Replaces one or many matches with a string |

## re.findall()

The re.findall() method returns a list of strings containing all matches.

### Example 1: re.findall()

# Program to extract numbers from a string

import re

string = 'hello 12 hi 89. Howdy 34'

pattern = '\d+'

result = re.findall(pattern, string)

print(result)

# Output: ['12', '89', '34']

### Example

import re  
txt = "The rain in Spain"  
x = re.findall("ai", txt)  
print(x)

o/p:

['ai', 'ai']

## re.split()

The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

### Example 2: re.split()

import re

string = 'Twelve:12 Eighty nine:89.'

pattern = '\d+'

result = re.split(pattern, string)

print(result)

# Output: ['Twelve:', ' Eighty nine:', '.']

## re.sub()

The syntax of re.sub() is:

re.sub(pattern, replace, string)

The method returns a string where matched occurrences are replaced with the content of replace variable.

### Example 3: re.sub()

# Program to remove all whitespaces

import re

# multiline string

string = 'abc 12\

de 23 \n f45 6'

# matches all whitespace characters

pattern = '\s+'

# empty string

replace = '~'

new\_string = re.sub(pattern, replace, string)

print(new\_string)

o/p:

abc~12de~23~f45~6

If the pattern is no found, re.sub() returns the original string.

## re.search()

The re.search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.

If the search is successful, re.search() returns a match object; if not, it returns None.

match = re.search(pattern, str)

### Ex

import re

string = "Python is fun"

# check if 'Python' is at the beginning

match = re.search('\APython', string)

if match:

print("pattern found inside the string")

else:

print("pattern not found")

# Output: pattern found inside the string

Ex:

import re  
txt = "The rain in Spain"  
x = re.search("Portugal", txt)  
print(x)

o/p:

None

ex:

import re

txt = "The rain in Spain"

x = re.search("Spain", txt)

print(x)

o/p:

<re.Match object; span=(12, 17), match='Spain'>

## Metacharacters

Metacharacters are characters with a special meaning:

|  |  |  |
| --- | --- | --- |
| **Character** | **Description** | **Example** |
| [] | A set of characters | **"[a-m]"** |

|  |  |  |
| --- | --- | --- |
| \ | Signals a special sequence (can also be used to escape special characters) | "\d" |
| . | Any character (except newline character) | "he..o" |
| ^ | Starts with | "^hello" |
| $ | Ends with | "world$" |
| \* | Zero or more occurrences | "aix\*" |
| + | One or more occurrences | "aix+" |
| {} | Exactly the specified number of occurrences | "al{2}" |
| | | Either or | "falls|stays" |
| () | Capture and group |  |

## Special Sequences

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

|  |  |  |  |
| --- | --- | --- | --- |
| **Character** | **Description** | **Example** | **Try it** |
| \A | Returns a match if the specified characters are at the beginning of the string | "\AThe" |  |
| \b | Returns a match where the specified characters are at the beginning or at the end of a word | r"\bain" r"ain\b" |  |
| \B | Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word | r"\Bain" r"ain\B" | \B |
| \d | Returns a match where the string contains digits (numbers from 0-9) | "\d" | \d |
| \D | Returns a match where the string DOES NOT contain digits | "\D" | \D |
| \s | Returns a match where the string contains a white space character | "\s" | \s |
| \S | Returns a match where the string DOES NOT contain a white space character | "\S" | \S |
| \w | Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore \_ character) | "\w" | \w |
| \W | Returns a match where the string DOES NOT contain any word characters | "\W" | \W |
| \Z | Returns a match if the specified characters are at the end of the string | "Spain\Z" | \Z |

## Sets

A set is a set of characters inside a pair of square brackets [] with a special meaning:

|  |  |  |  |
| --- | --- | --- | --- |
| **Character** | **Description** | **Example** | **Try it** |
| [arn] | Returns a match where one of the specified characters (a, r, or n) are present |  |  |
| [a-n] | Returns a match for any lower case character, alphabetically between a and n |  |  |
| [^arn] | Returns a match for any character EXCEPT a, r, and n |  |  |
| [0123] | Returns a match where any of the specified digits (0, 1, 2, or 3) are present |  |  |
| [0-9] | Returns a match for any digit between 0 and 9 |  |  |
| [0-5][0-9] | Returns a match for any two-digit numbers from 00 and 59 |  |  |
| [a-zA-Z] | Returns a match for any character alphabetically between a and z, lower case OR upper case |  |  |
| [+] | In sets, +, \*, ., |, (), $,{} has no special meaning, so [+] means: return a match for any + character in the string |  |  |

**Match():**

It contains: span(), string(), group()

span() : It returns the tuple containing the starting and end position of the match.

string() : it returns a string passed into the function

group() : the part of the string is returned where the match is found.

**ITERATOR:**

Python Iterators are objects that can be iterated upon.

What are iterators in Python?

Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but hidden in plain sight. Iterator in Python is simply an [object](https://www.programiz.com/python-programming/class) that can be iterated upon.

Python **iterator object** must implement two special methods, \_\_iter\_\_() and \_\_next\_\_(), collectively called the **iterator protocol**.

An object is called **iterable** if we can get an iterator from it. Most of built-in containers in Python like: [list](https://www.programiz.com/python-programming/list), [tuple](https://www.programiz.com/python-programming/tuple), [string](https://www.programiz.com/python-programming/string) etc. are iterables.

**Example1:**

mytuple = ("apple", "banana", "cherry")

myit = iter(mytuple)

print(next(myit))

print(next(myit))

print(next(myit))

o/p:

apple

banana

cherry

**example2:**

Strings are also iterable objects, containing a sequence of characters:

mystr = "banana"  
myit = iter(mystr)  
  
print(next(myit))  
print(next(myit))  
print(next(myit))  
print(next(myit))  
print(next(myit))  
print(next(myit))

o/p:

b

a

n

a

n

a

## Looping Through an Iterator

We can also use a for loop to iterate through an iterable object:

### Example1: Iterate the values of a tuple:

mytuple = ("apple", "banana", "cherry")  
for x in mytuple:  
  print(x)

o/p:

apple

banana

cherry

The \_\_iter\_\_() method acts similarly, we can do same operations , but it must always return the iterator object itself.

**Example2:** Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

class MyNumbers:  
  def \_\_iter\_\_(self):  
    self.a = 1  
    return self  
  def \_\_next\_\_(self):  
    x = self.a  
    self.a += 1  
    return x  
  
myobj= MyNumbers()  
myiter = iter(myobj)  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))  
print(next(myiter))

o/p:

1

2

3

4

5

**Example3:**

class MyNumbers:

def \_\_iter\_\_(self):

self.a = 1

return self

def \_\_next\_\_(self):

x = self.a

self.a += 1

return x

myobj= MyNumbers()

myiter = iter(myobj)

for i in range(10):

print(next(myiter), end=",")

o/p:

1,2,3,4,5,6,7,8,9,10,

## \*\*\*\*\*\*StopIteration

To prevent the iteration, we can use the StopIteration statement.

In the \_\_next\_\_() method, we can add a terminating condition to raise an error.

Example1:without StopIteration

class MyNumbers:

def \_\_iter\_\_(self):

self.a = 1

return self

def \_\_next\_\_(self):

if self.a<=5:

x = self.a

self.a += 1

return x

myobj= MyNumbers()

myiter = iter(myobj)

for i in range(10):

print(next(myiter), end=",")

o/p:

1,2,3,4,5,None,None,None,None,None,

### Example2:

Stop after 10 iterations:

class MyNumbers:

def \_\_iter\_\_(self):

self.a = 1

return self

def \_\_next\_\_(self):

if self.a <= 10:

x = self.a

self.a += 1

return x

else:

raise StopIteration

myclass = MyNumbers()

myiter = iter(myclass)

for x in myiter:

print(x,end=" ")

o/p:

1 2 3 4 5 6 7 8 9 10

**GENERATOR**

Generators are the functions that return the traversal object and used to create iterators. It traverses the entire items at once.

There is a lot of complexity in creating iteration in Python; we need to implement **\_\_iter\_\_()** and **\_\_next\_\_()** method to keep track of internal states. It is a lengthy process to create iterators.

That's why the generator plays an essential role in simplifying this process. If there is no value found in iteration, it raises **StopIteration** exception.

## How to Create Generator function in Python?

It is similar to the normal function defined by the **def** keyword and uses a **yield** keyword instead of return.

Or we can say that if the body of any function contains a **yield** statement, it automatically becomes a generator function.

Example: Write a program to print the table of the given number using the generator.

def table(n):

for i in range(1,11):

yield n\*i

i = i+1

for x in table(15):

print(x)

o/p:

15

30

45

60

75

90

105

120

135

150

**yield vs. return**

The **yield** statement is responsible for controlling the flow of the generator function.

It pauses the function execution by saving all states and yielded to the caller. Later it resumes execution when a successive function is called.

We can use the multiple yield statement in the generator function.

The return statement **returns** a value and terminates the whole function and only one return statement can be used in the function.

**Using multiple yield Statement:**

We can use the multiple yield statement in the generator function. Consider the following example.

def multiple\_yield():

    str1 = "First String"

    yield str1

    str2 = "Second string"

    yield str2

    str3 = "Third String"

    yield str3

obj = multiple\_yield()

print(next(obj))

print(next(obj))

print(next(obj))

**Output:**

First String

Second string

Third String

Ex2: # A simple generator function

def my\_gen():

n = 1

print('This is printed first')

# Generator function contains yield statements

yield n

n += 1

print('This is printed second')

yield n

n += 1

print('This is printed at last')

yield n

# Using for loop

for item in my\_gen():

print(item)

o/p:

This is printed first

1

This is printed second

2

This is printed at last

3

## Python Generators with a Loop

Normally, generator functions are implemented with a loop having a suitable terminating condition.

an example of a generator that reverses a string.

def rev\_str(my\_str):

length = len(my\_str)

for i in range(length-1,-1,-1):

yield my\_str[i]

# For loop to reverse the string

for c in rev\_str("hello"):

print(c)

o/p:

o

l

l

e

h

**Difference between Generator function and Normal function**

* Normal function contains only one **return** statement whereas generator function can contain one or more **yield** statement.
* When the generator functions are called, the normal function is paused immediately and control transferred to the caller.
* Local variable and their states are remembered between successive calls.
* **StopIteration exception** is raised automatically when the function terminates.

**Example: Advantage of Generator compare to Iterator**

Generators can be implemented in a clear and concise way as compared to their iterator class.

Following is an example to implement a sequence of power of 2's using iterator class.

class powtwoiter:

def \_\_init\_\_(self, max = 0):

self.max = max

def \_\_iter\_\_(self):

self.n = 0

return self

def \_\_next\_\_(self):

if self.n > self.max:

raise StopIteration

result = 2 \*\* self.n

self.n += 1

return result

x=powtwoiter(3)

myit=iter(x)

print(next(myit))

print(next(myit))

o/p:

1

2

Now lets do the same using a generator function.

def powtwogen(max = 0):

n = 0

while n < max:

yield 2 \*\* n

n += 1

x=powtwogen(3)

myit=iter(x)

print(next(myit))

print(next(myit))

o/p:

1

2

## Python Generator Expression

Simple generators can be easily created on the fly using generator expressions. It makes building generators easy.

Same as lambda function creates an [anonymous function](https://www.programiz.com/python-programming/anonymous-function), generator expression creates an anonymous generator function.

The syntax for ***generator expression*** is similar to that of a [***list comprehension*** in Python](https://www.programiz.com/python-programming/list#list-comprehension). But the square brackets are replaced with round parentheses.

**Example of list comprehension:**

List comprehension consists of an expression followed by [for statement](https://www.programiz.com/python-programming/for-loop) inside square brackets.

Here is an example to make a list with each item being increasing power of 2.

pow2 = [2 \*\* x for x in range(10)]

# Output: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]

print(pow2)

This code is equivalent to

pow2 = []

for x in range(10):

pow2.append(2 \*\* x)

A list comprehension can optionally contain more for or [if statements](https://www.programiz.com/python-programming/if-elif-else). An optional if statement can filter out items for the new list. Here are some examples.

>>> pow2 = [2 \*\* x for x in range(10) if x > 5]

>>> pow2

[64, 128, 256, 512]

>>> odd = [x for x in range(20) if x % 2 == 1]

>>> odd

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

***The major difference between a list comprehension and a generator expression is that while list comprehension produces the entire list, generator expression produces one item at a time.***

Example1:

# Initialize the list

my\_list = [1, 3, 6, 10]

# square each term using list comprehension

# Output: [1, 9, 36, 100]

print([x\*\*2 for x in my\_list])

# same thing can be done using generator expression

# Output: <generator object <genexpr> at 0x0000000002EBDAF8>

# Initialize the list

a = (x\*\*2 for x in my\_list)

# Output: 1

print(next(a))

# Output: 9

print(next(a))

# Output: 36

print(next(a))

# Output: 100

print(next(a))

# Output: StopIteration

next(a)

# **Python Decorator**

"In Decorators, functions are passed as an argument into another function and then called inside the wrapper function."

It is also called **meta programming** where a part of the program attempts to change another part of program at compile time.

Before we go for decorator, ffirst you have to know or remember that function names are references to functions and that we can assign multiple names to the same function:

Ex1:

def func1(msg):

print(msg)

func1("Hii")

func2 = func1

func2("Hii")

del func1

func1(“hello”) #error

func2(“hello”)

o/p:

Hii

Hii

Ex2:

>>> def succ(x):

... return x + 1

...

>>> successor = succ

>>> successor(10)

11

>>> succ(10)

11

In the above program, we need to understand the following concept of the function:

* The function can be referenced and passed to a variable and returned from other functions as well.
* The functions can be declared inside another function and passed as an argument to another function.

## Inner Function

Python provides the facility to define the function inside another function. These types of functions are called inner functions. Consider the following example:

def func():

     print("We are in first function")

     def func1():

           print("This is first child function")

     def func2():

           print(" This is second child function")

     func1()

     func2()

func()

**Output:**

We are in first function

This is first child function

This is second child function

function that accepts other function as an argument is also called **higher order function**. Consider the following example:

def add(x):

return x+1

def sub(x):

return x-1

def operator(func, x):

temp = func(x)+x

return temp

print(operator(sub,10))

print(operator(add,20))

**Output:**

19

41

Python decorators allows us to decorate a function.

It allows you to take on extra functionality to an existing function.

Ex:

def simple\_function():

#do something

To add more functionality to this function if we require, then there is a problem.

Here we have two options:

* 1. Add extra code to the old function
  2. Create a new function by copying the old code and then adding new code to that.

But again if we want to remove the extra functionality then, we need to delete them manually or check for the old function if it is exist.

If we have some ON/OFF switch for adding functionality to the existing function then it is called decorator.

For it we use the @ operator at the top of the original function.

Example:

@some\_decorator

def simple\_function():

#dop something

Now let us go through the steps of manually building a decorator to know what @ operator is doing behind the scene.

Ex1:

def func():

return 1;

func()

o/p:

1

If we type:

func

then the error displayed as:

< function\_\_main\_\_.func>

So, it is showing error message that we have a function func

Ex2:

def hello():

return “hello”

hello()

o/p:

hello

if we type:

greet=hello

greet()

o/p:

hello

* Here actually greet is pointing to the function object hello().
* Here even we delete hello(), then also the greet() will be exist and will work.

Ex:

del hello

hello() # it will error

greet()

o/p:

hello

**define a function inside another.**

def f():

def g():

print("Hi, it's me 'g'")

print("Thanks for calling me")

print("This is the function 'f'")

print("I am calling 'g' now:")

g()

f()

We will get the following output, if we start the previous program:

This is the function 'f'

I am calling 'g' now:

Hi, it's me 'g'

Thanks for calling me

Another example using "proper" return statements in the functions:

def temperature(t):

def celsius2fahrenheit(x):

return 9 \* x / 5 + 32

result = "It's " + str(celsius2fahrenheit(t)) + " degrees!"

return result

print(temperature(20))

The output:

It's 68.0 degrees!

**passing a function into another**

we can pass functions - or better "references to functions" - as parameters to a function. We will demonstrate this in the next simple example:

def g():

print("Hi, it's me 'g'")

print("Thanks for calling me")

def f(func):

print("Hi, it's me 'f'")

print("I will call 'func' now")

func()

f(g)

The output looks like this:

Hi, it's me 'f'

I will call 'func' now

Hi, it's me 'g'

Thanks for calling me

def g():

print("Hi, it's me 'g'")

print("Thanks for calling me")

def f(func):

print("Hi, it's me 'f'")

print("I will call 'func' now")

func()

print("func's real name is " + func.\_\_name\_\_)

f(g)

The output explains once more what's going on:

Hi, it's me 'f'

I will call 'func' now

Hi, it's me 'g'

Thanks for calling me

func's real name is g

**EXAMPLE USING PYTHON DECORATOR:**

Ex1:consider a function and observe the output

def newfun(original):

def func():

print("extra code before");

original();

print("extra code after");

return func

def original():

print("I m original")

original()

o/p:

I m original

Ex2: observe the 2nd output

def newfun(original):

def func():

print("extra code before");

original();

print("extra code after");

return func

def original():

print("I m original")

d=newfun(original)

d()

o/p:

extra code before

I m original

extra code after

Now the function which needed to be decorated i.e: original() can be modified by prefix one statement as: @newfun

Ex3:

def newfun(original):

def func():

print("extra code before");

original();

print("extra code after");

return func

@newfun

def original():

print("I m original")

original() # d=newfun(original)

o/p:

extra code before

I m original

extra code after

Note: When we call the original() function actually this function is supplied as argument to the newfun(), as because we have written @followed by decorator function newfun.

So, now we can go for switch ON or OFF to the statement written with @

Eg:

@newfun

Or

#@newfun

Decorators are mainly used in web frame work applications and other software development fields.