# Python Functions

# Python Lambda Functions

# Python File Handling

# Python Modules

# Python Exceptions

# Python Date and time

# Python Sending Email using SMTP

# Python read csv file

# Python read excel file

# Python List Comprehension

# Python Math Module

# Python OS Module

# Python Random module

# Python statistics module

# Python sys module

# 

# 

# 

# 

# 

# 

# 

# Python Functions

## Advantages of Functions in Python

* By including functions, we can prevent repeating the same code block repeatedly in a program.
* Python functions, once defined, can be called many times and from anywhere in a program.
* If our Python program is large, it can be separated into numerous functions which are simple to track.
* The key accomplishment of Python functions is we can return as many outputs as we want with different arguments.

However, calling functions has always been overhead in a Python program.

**def** function\_name( parameters ):

# code block

1. **def** square( num ): ex:-# Example Python Code for User-Defined function
2. **return** num\*\*2
3. object\_ = square(6)
4. **print**( "The square of the given number is: ", object\_ )

## Pass by Reference vs. Pass by Value

All parameters in the Python programming language are provided by reference. It indicates that if we alter the value of an argument inside of a function, the calling function will likewise reflect the change.

1. **def** square( item\_list ): # Example Python Code for Pass by Reference vs. Value
2. squares = [ ]
3. **for** l **in** item\_list:
4. squares.append( l\*\*2 )
5. **return** squares
6. my\_list = [17, 52, 8];
7. my\_result = square( my\_list )
8. **print**( "Squares of the list are: ", my\_result ) // [289, 2704, 64]

## Function Arguments

The following are the types of arguments that we can use to call a function:

1. Default arguments
2. Keyword arguments
3. Required arguments
4. Variable-length arguments

### 1) Default Arguments

A default argument is a kind of parameter that takes as input a default value if no value is supplied for the argument when the function is called. Default arguments are demonstrated in the following instance.

1. **def** function( n1, n2 = 20 ):
2. **print**("number 1 is: ", n1)
3. **print**("number 2 is: ", n2)
4. # Calling the function and passing only one argument
5. function(30) //30,20
6. # Now giving two arguments to the function
7. function(50,30) //50,30

### 2) Keyword Arguments

A function called's arguments are linked to keyword arguments. When invoking a function with keyword arguments, the user may tell whose parameter value it is by looking at the parameter label.

We can remove certain arguments or arrange them in a different order since the Python interpreter will connect the provided keywords to link the values with its parameters. Another way to use keywords to invoke the function() method is as follows:

1. **def** function( n1, n2 ):
2. **print**("number 1 is: ", n1)
3. **print**("number 2 is: ", n2)
4. # Calling function and passing arguments without using keyword
5. function( 50, 30) //50,30
6. # Calling function and passing arguments using keyword
7. function( n2 = 50, n1 = 30) //30,50

### 3) Required Arguments

The arguments given to a function while calling in a pre-defined positional sequence are required arguments. The count of required arguments in the method call must be equal to the count of arguments provided while defining the function.

1. **def** function( n1, n2 ):
2. **print**("number 1 is: ", n1)
3. **print**("number 2 is: ", n2)
4. # Calling function and passing two arguments out of order, we need num1 to be 20 and num2 to be 30
5. function( 30, 20 ) //30,20
6. **try**: # Calling function and passing only one argument
7. function( 30 ) //error need two arguments
8. **except**:
9. **print**( "Function needs two positional arguments" )

### 4) Variable-Length Arguments

We can use special characters in Python functions to pass as many arguments as we want in a function. There are two types of characters that we can use for this purpose:

* **\*args -**These are Non-Keyword Arguments
* **\*\*kwargs -**These are Keyword Arguments.

1. **def** function( \*args\_list ):
2. ans = []
3. **for** l **in** args\_list:
4. ans.append( l.upper() )
5. **return** ans
6. object = function('Python', 'Functions', 'tutorial') # Passing args arguments
7. **print**( object ) //['PYTHON', 'FUNCTIONS', 'TUTORIAL']
8. **def** function( \*\*kargs\_list ): # defining a function
9. ans = []
10. **for** key, value **in** kargs\_list.items():
11. ans.append([key, value])
12. **return** ans
13. # Paasing kwargs arguments
14. object = function(First = "Python", Second = "Functions", Third = "Tutorial")
15. **print**(object) //[['First', 'Python'], ['Second', 'Functions'], ['Third', 'Tutorial']]

## return Statement

We write a return statement in a function to leave a function and give the calculated value when a defined function is called.

## The Anonymous Functions

These types of Python functions are anonymous since we do not declare them, as we declare usual functions, using the def keyword. We can use the lambda keyword to define the short, single output, anonymous functions.

Lambda expressions can accept an unlimited number of arguments; however, they only return one value as the result of the function. They can't have numerous expressions or instructions in them. Since lambda needs an expression, an anonymous function cannot be directly called to print.

**lambda** [argument1 [,argument2... .argumentn]] : expression

1. # Python code to demonstrate ananymous functions
2. lambda\_ = **lambda** argument1, argument2: argument1 + argument2;
3. **print**( "Value is : ", lambda\_( 20, 30 ) ) //50 # Calling function & passing values

## Scope and Lifetime of Variables

The scope of a variable refers to the domain of a program wherever it is declared. A function's arguments and variables are not accessible outside the defined function. As a result, they only have a local domain.

lifespan of a variable in RAM is how long it stays there. A function's lifespan is same as that of its internal variables. They are taken away after we exit function. Consequently, a function does not keep the value of a variable from previous executions.

1. **def** number( ):
2. num = 50
3. **print**( "Value of num inside the function: ", num) //50
4. num = 10
5. number()
6. **print**( "Value of num outside the function:", num) //10

## Python Function within Another Function

Inner or nested function refers to a function defined within another defined function. Inner functions can access the parameters of the outer scope. Inner functions are constructed to cover them from the changes that happen outside the function. **Many developers regard this process as encapsulation.**

1. **def** word():
2. string = 'Python functions tutorial'
3. x = 5
4. **def** number():
5. **print**( string )
6. **print**( x )
7. number()
8. word()

**Output:**Python functions tutorial 5

# Python Built-in Functions

The functions whose functionality is pre-defined in Python.

## Python abs() Function

The python **abs()** function is used to return the absolute value of a number. It takes only one argument. The argument can be an integer and floating-point number. If the argument is a complex number, then, abs() returns its magnitude.

1. integer = -20 # integer number
2. **print**('Absolute value of -20 is:', abs(integer)) //20
3. floating = -20.83 # floating number
4. **print**('Absolute value of -20.83 is:', abs(floating)) //20.83

## Python all() Function

The python **all()** function accepts an iterable object (such as list, dictionary, etc.). It returns true if all items in passed iterable are true. Otherwise, it returns False. **If the iterable object is empty, the all() function returns True.**

1. k = [1, 3, 4, 6] # all values true
2. **print**(all(k)) //True
3. k = [0, False] # all values false
4. **print**(all(k)) //False
5. k = [1, 3, 7, 0] # one false value
6. **print**(all(k)) //False
7. k = [0, False, 5] # one true value
8. **print**(all(k)) //False
9. k = [] # empty iterable
10. **print**(all(k)) //True

## Python bin() Function

The python **bin()** function is used to return the binary representation of a specified integer. A result always starts with the prefix 0b.

1. x = 10
2. y = bin(x)
3. **print** (y) //0b1010

## Python bool()

The python **bool()** converts a value to boolean(True or False) using the standard truth testing procedure.

1. test1 = []
2. **print**(test1,'is',bool(test1)) //False
3. test1 = [0]
4. **print**(test1,'is',bool(test1)) //True
5. test1 = 0.0
6. **print**(test1,'is',bool(test1)) //False
7. test1 = None
8. **print**(test1,'is',bool(test1)) //False
9. test1 = True
10. **print**(test1,'is',bool(test1)) //True
11. test1 = 'Easy string'
12. **print**(test1,'is',bool(test1)) //True

## Python bytes()

The python **bytes()** in Python is used for returning a **bytes** object. It is an immutable version of the bytearray() function.

It can create empty bytes object of the specified size.

1. string = "Hello World."
2. array = bytes(string, 'utf-8')
3. **print**(array) //b ' Hello World.'

## Python callable() Function

A python **callable()** function in Python is something that can be called. This built-in function checks & returns true if object passed appears to be callable, otherwise false.

1. x = 8
2. **print**(callable(x)) //False

## Python compile() Function

The python **compile()** function takes source code as input and returns a code object which can later be executed by exec() function.

1. # compile string source to code
2. code\_str = 'x=5\ny=10\nprint("sum =",x+y)'
3. code = compile(code\_str, 'sum.py', 'exec')
4. **print**(type(code)) //<class 'code'>
5. **exec**(code) //sum = 15
6. **exec**(x)

## Python exec() Function

The python **exec()** function is used for the dynamic execution of Python program which can either be a string or object code and it accepts large blocks of code, unlike the eval() function which only accepts a single expression.

1. x = 8
2. **exec**('print(x==8)') //True
3. **exec**('print(x+4)') //12

## Python sum() Function

python **sum()** function is used to get the sum of numbers of an iterable, i.e., list.

1. s = sum([1, 2,4 ])
2. **print**(s) // 7
3. s = sum([1, 2, 4], 10)
4. **print**(s) // 17

## Python any() Function

Python **any()** function returns true if any item in an iterable is true. else, it returns False.

1. l = [4, 3, 2, 0]
2. **print**(any(l)) #True
3. l = [0, False]
4. **print**(any(l)) #False
5. l = [0, False, 5] #True
6. **print**(any(l))
7. l = []
8. **print**(any(l)) #False

## Python ascii() Function

The python **ascii()** function returns a string containing a printable representation of an object and escapes the non-ASCII characters in the string using \x, \u or \U escapes.

1. normalText = 'Python is interesting'
2. **print**(ascii(normalText)) // 'Python is interesting'
3. otherText = 'Pythön is interesting'
4. **print**(ascii(otherText)) //'Pyth\xf6n is interesting'
5. **print**('Pyth\xf6n is interesting') //Pythön is interesting

## Python bytearray()

The python **bytearray()** returns a bytearray object and can convert objects into bytearray objects, or create an empty bytearray object of the specified size.

1. string = "Python is a programming language."
2. arr = bytearray(string, 'utf-8') # string with encoding 'utf-8'
3. **print**(arr) //bytearray(b'Python is a programming language.')

## Python eval() Function

The python **eval()** function parses the expression passed to it and runs python expression(code) within the program.

1. x = 8
2. **print**(eval('x + 1')) **//9**

## Python format() Function

The python **format()** function returns a formatted representation of the given value.

1. **print**(format(123, "d")) **//123**
2. **print**(format(123.4567898, "f")) **//123.456790**
3. **print**(format(12, "b")) **//1100**

## Python frozenset()

The python **frozenset()** function returns an immutable frozenset object initialized with elements from the given iterable.

## Python getattr() Function

The python **getattr()** function returns the value of a named attribute of an object. If it is not found, it returns the default value.

1. **class** Details:
2. age = 22
3. name = "Phill"
4. details = Details()
5. **print**('The age is:', getattr(details, "age")) //22
6. **print**('The age is:', details.age) //22

## Python globals() Function

The python **globals()** function returns the dictionary of the current global symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

1. age = 22
2. globals()['age'] = 22
3. **print**('The age is:', age) //22

## Python locals() Function

Python **locals()** method updates & returns the dictionary of current local symbol table.

A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes, etc.

1. **def** localsAbsent():
2. **return** locals()
3. **def** localsPresent():
4. present = True
5. **return** locals()
6. **print**('localsNotPresent:', localsAbsent()) //localsAbsent: {}
7. **print**('localsPresent:', localsPresent()) //localsPresent: {'present': True}

## Python iter() Function

The python **iter()** function is used to return an iterator object. It creates an object which can be iterated one element at a time.

1. list = [1,2,3]
2. listIter = iter(list)
3. **print**(next(listIter)) //1
4. **print**(next(listIter)) //2
5. **print**(next(listIter)) //3

## Python list(): The python **list()** creates a list in python.

1. **print**(list()) //[]
2. String = 'abcde'
3. **print**(list(String)) //['a', 'b', 'c', 'd', 'e']
4. Tuple = (1,2,3,4,5)
5. **print**(list(Tuple)) //[1,2,3,4,5]
6. List = [1,2,3,4,5]
7. **print**(list(List)) //[1,2,3,4,5]

## Python map() Function

The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

The python **map()** function is used to return a list of results after applying a given function to each item of an iterable(list, tuple etc.).

1. **def** calculateAddition(n):
2. **return** n+n
3. numbers = (1, 2, 3, 4)
4. result = map(calculateAddition, numbers)
5. **print**(result) //<map object at 0x7fb04a6bec18>
6. numbersAddition = set(result) # converting map object to set
7. **print**(numbersAddition) //{8, 2, 4, 6}

## Python object()

The python **object()** returns an empty object. It is a base for all the classes and holds the built-in properties and methods which are default for all the classes.

1. python = object()
2. **print**(type(python)) //<class 'object'>

## Python open() Function

The python **open()** function opens the file and returns a corresponding file object.

1. f = open("python.txt")
2. f = open("C:/Python33/README.txt") # specifying full path

## Python chr() Function

Python **chr()** function is used to get a string representing a character which points to a Unicode code integer. For example, chr(97) returns the string 'a'. This function takes an integer argument and throws an error if it exceeds the specified range. The standard range of the argument is from 0 to 1,114,111.

## Python complex()

Python **complex()** function is used to convert numbers or string into a complex number. This method takes two optional parameters and returns a complex number. The first parameter is called a real and second as imaginary parts.

1. a = complex(1) # Passing single parameter
2. b = complex(1,2) # Passing both parameters
3. **print**(a) //(1.5+0j)
4. **print**(b) ////(1.5+2.2j)

## Python delattr() Function

Python **delattr()** function is used to delete an attribute from a class. It takes two parameters, first is an object of the class and second is an attribute which we want to delete. After deleting the attribute, it no longer available in the class and throws an error if try to call it using the class object.

## Python dir() Function

Python **dir()** function returns the list of names in the current local scope. If the object on which method is called has a method named \_\_dir\_\_(), this method will be called and must return the list of attributes. It takes a single object type argument.

1. att = dir()
2. **print**(att)

## Python divmod() Function

Python **divmod()** function is used to get remainder and quotient of two numbers. It takes two arguments and returns a tuple. Both arguments are required and numeric

1. result = divmod(10,2)
2. **print**(result) //(5,0)

## Python enumerate() Function

Python **enumerate()** function returns an enumerated object. It takes two parameters, first is a sequence of elements and the second is the start index of the sequence. We can get the elements in sequence either through a loop or next() method.

1. result = enumerate([1,2,3])
2. **print**(result) //<enumerate object at 0x7ff641093d80>
3. **print**(list(result)) //[(0, 1), (1, 2), (2, 3)]

## Python dict()

Python **dict()** function is a constructor which creates a dictionary. Python dictionary provides three different constructors to create a dictionary:

* If no argument is passed, it creates an empty dictionary.
* If a positional argument is given, a dictionary is created with the same key-value pairs. Otherwise, pass an iterable object.
* If keyword arguments are given, the keyword arguments and their values are added to the dictionary created from the positional argument.

1. result = dict() # returns an empty dictionary
2. result2 = dict(a=1,b=2)
3. **print**(result) ///{}
4. **print**(result2) //{'a': 1, 'b': 2}

## Python filter() Function

Python **filter()** function is used to get filtered elements. This function takes two arguments, first is a function and the second is iterable. The filter function returns a sequence of those elements of iterable object for which function returns **true value**.

The first argument can be **none**, if the function is not available and returns only elements that are **true**.

1. **def** filterdata(x):
2. **if** x>5:
3. **return** x
4. result = filter(filterdata,(1,2,6))
5. **print**(list(result)) //[6]

## Python hash() Function

Python **hash()** function is used to get the hash value of an object. Python calculates the hash value by using the hash algorithm. The hash values are integers and used to compare dictionary keys during a dictionary lookup. We can hash only the types which are given below:

**Hashable types:** \* bool \* int \* long \* float \* string \* Unicode \* tuple \* code object.

1. result = hash(21) # integer value
2. result2 = hash(22.2) # decimal value
3. **print**(result) //21
4. **print**(result2) //461168601842737174

## Python min() Function

Python **min()** function is used to get the smallest element from the collection. This function takes two arguments, first is a collection of elements and second is key, and returns the smallest element from the collection.

## Python set() Function

It is used to create a new set using elements passed during the call. It takes an iterable object as an argument and returns a new set object.

1. result = set() # empty set
2. result2 = set('12')
3. result3 = set('javatpoint')
4. **print**(result) //set()
5. **print**(result2) ///{‘1’,’2’}
6. **print**(result3) //{'a', 'n', 'v', 't', 'j', 'p', 'i', 'o'}

## Python hex() Function

Python **hex()** function is used to generate hex value of an integer argument. It takes an integer argument and returns an integer converted into a hexadecimal string. In case, we want to get a hexadecimal value of a float, then use float.hex() function.

1. result2 = hex(342)
2. **print**(result2) //0x156

## Python id() Function

Python **id()** function returns the identity of an object. This is an integer which is guaranteed to be unique. This function takes an argument as an object and returns a unique integer number which represents identity. Two objects with non-overlapping lifetimes may have the same id() value.

1. val = id("Javatpoint") # string object
2. val2 = id(1200) # integer object
3. val3 = id([25,336,95,236,92,3225]) # List object
4. **print**(val) //139963782059696
5. **print**(val2) //139963805666864
6. **print**(val3) /139963781994504

## Python setattr() Function

Python **setattr()** function is used to set a value to the object's attribute. It takes three arguments, i.e., an object, a string, and an arbitrary value, and returns none. It is helpful when we want to add a new attribute to an object and set a value to it.

## Python slice() Function

Python **slice()** function is used to get a slice of elements from the collection of elements. Python provides two overloaded slice functions. The first function takes a single argument while the second function takes three arguments and returns a slice object. This slice object can be used to get a subsection of the collection.

1. result = slice(5) # returns slice object
2. result2 = slice(0,5,3) # returns slice object
3. **print**(result) //slice(None, 5, None)
4. **print**(result2) //slice(0, 5, 3)

## Python sorted() Function

Python **sorted()** function is used to sort elements. By default, it sorts elements in an ascending order but can be sorted in descending also. It takes four arguments and returns a collection in sorted order. In case of a dictionary, it sorts only keys, not values.

1. str = "javatpoint" # declaring string
2. sorted1 = sorted(str) # sorting string
3. **print**(sorted1) /**/['a', 'a', 'i', 'j', 'n', 'o', 'p', 't', 't', 'v']**

## Python next() Function

Python **next()** function is used to fetch next item from the collection. It takes two arguments, i.e., an iterator and a default value, and returns an element.

This method calls on iterator and throws an error if no item is present. To avoid the error, we can set a default value.

1. number = iter([256, 32, 82]) # Creating iterator
2. item = next(number)
3. **print**(item) //256
4. item = next(number) //# second item
5. **print**(item) //32
6. item = next(number) //# third item
7. **print**(item) //82

## Python input() Function

Python **input()** function is used to get an input from the user. It prompts for the user input and reads a line. After reading data, it converts it into a string and returns it. It throws an error **EOFError** if EOF is read.

1. val = input("Enter a value: ")
2. **print**("You entered:",val)

## Python isinstance() Function

Python **isinstance()** function is used to check whether the given object is an instance of that class. If the object belongs to the class, it returns true. Otherwise returns False. It also returns true if the class is a subclass.

The **isinstance()** function takes two arguments, i.e., object and classinfo, and then it returns either True or False.

## Python oct() Function

Python **oct()** function is used to get an octal value of an integer number. This method takes an argument and returns an integer converted into an octal string. It throws an error **TypeError**, if argument type is other than an integer.

1. val = oct(10)
2. **print**("Octal value of 10:",val) //0o12

## Python ord() Function

The python **ord()** function returns an integer representing Unicode code point for the given Unicode character.

**print**(ord('8')) //56

## Python pow() Function

Python **pow()** method is used to compute power of a number. It returns x to the power of y. If third argument(z) is given, it returns x to the power of y modulus z, i.e. (x, y) % z.

1. **print**(pow(4, 2)) //16
2. **print**(pow(-4, 2)) //16
3. **print**(pow(4, -2)) //0.0625
4. **print**(pow(-4, -2)) //0.0625

## Python range() Function

The python **range()** function returns an immutable sequence of numbers starting from 0 by default, increments by 1 (by default) and ends at a specified number.

1. **print**(list(range(0))) **//[ ]**
2. **print**(list(range(4)))  **//[0, 1, 2, 3]**
3. **print**(list(range(1,7 ))) **//[1, 2, 3, 4, 5, 6]**

## Python reversed() Function

The python **reversed()** function returns the reversed iterator of the given sequence.

1. String = 'Java'
2. **print**(list(reversed(String))) **//['a', 'v', 'a', 'J']**
3. Tuple = ('J', 'a', 'v', 'a')
4. **print**(list(reversed(Tuple)) **//['a', 'v', 'a', 'J']**
5. Range = range(8, 12)
6. **print**(list(reversed(Range))  **//[11, 10, 9, 8]**
7. List = [1, 2, 7, 5]
8. **print**(list(reversed(List)))  **//[5, 7, 2, 1]**

## Python round() Function

The python **round()** function rounds off the digits of a number and returns the floating point number.

1. **print**(round(10)) //10
2. **print**(round(10.8)) //11

## Python issubclass() Function

The python **issubclass()** function returns true if object argument(first argument) is a subclass of second class(second argument).

## Python tuple(): python **tuple()** function is used to create a tuple object.

1. t1 = tuple()
2. **print**('t1=', t1) **//t1= ()**
3. t2 = tuple([1, 6, 9])
4. **print**('t2=', t2) **//t2= (1, 6, 9)**
5. t1 = tuple('Java')
6. **print**('t1=',t1) **//t1= ('J', 'a', 'v', 'a')**
7. t1 = tuple({4: 'four', 5: 'five'})
8. **print**('t1=',t1) **//t1= (4, 5)**

## Python type()

Python **type()** returns the type of the specified object if a single argument is passed to type() built in function. If three arguments are passed, then it returns a new type object.

## Python vars() function

The python **vars()** function returns the \_\_dict\_\_ attribute of the given object.

1. **class** Python:
2. **def** \_\_init\_\_(self, x = 7, y = 9):
3. self.x = x
4. self.y = y
5. InstanceOfPython = Python()
6. **print**(vars(InstanceOfPython))  **//{'y': 9, 'x': 7}**

## Python zip() Function

The python **zip()** Function returns a zip object, which maps a similar index of multiple containers. It takes iterables (can be zero or more), makes it an iterator that aggregates the elements based on iterables passed, and returns an iterator of tuples.

1. numList = [4,5, 6]
2. strList = ['four', 'five', 'six']
3. result = zip()
4. resultList = list(result)
5. **print**(resultList) **//[ ]**
6. result = zip(numList, strList)
7. resultSet = set(result)
8. **print**(resultSet) **//{(5, 'five'), (4, 'four'), (6, 'six')}**

# Python Lambda Functions

Lambda Functions in Python are anonymous functions, implying they don't have a name. The def keyword is needed to create a typical function in Python, as we already know. We can also use the lambda keyword in Python to define an unnamed function.

**lambda** arguments: expression

This function accepts any count of inputs but only evaluates & returns one expression.

1. add = **lambda** num: num + 4
2. **print**( add(6) ) //10

Or

1. **def** add( num ):
2. **return** num + 4
3. **print**( add(6) ) //10

## Using Lambda Function with filter()

Filter() method accepts two arguments: a function and an iterable such as a list.

1. list\_ = [34, 12, 64, 55, 75, 13, 63] # Code to filter odd numbers from a given list
2. odd\_list = list(filter( **lambda** num: (num % 2 != 0) , list\_ ))
3. **print**(odd\_list) //[55, 75, 13, 63]

## Using Lambda Function with List Comprehension

## We'll apply lambda function combined with list comprehension & lambda keyword with a for loop in this instance. We'll attempt to print square of numbers in the range 0 to 11.

1. squares = [**lambda** num = num: num \*\* 2 **for** num **in** range(0, 11)]
2. **for** square **in** squares:
3. **print**( square(), end = " ") //0 1 4 9 16 25 36 49 64 81 100

## Using Lambda Function with if-else

We will use the lambda function with the if-else block.

1. Minimum = **lambda** x, y : x **if** (x < y) **else** y
2. **print**(Minimum( 35, 74 )) //35

## Using Lambda with Multiple Statements

Multiple expressions are not allowed in lambda functions, but we can construct 2 lambda functions or more and afterward call the second lambda expression as an argument to the first.

# Code to print the third-largest number of the given list using the lambda function

1. my\_List = [ [3, 5, 8, 6], [23, 54, 12, 87], [1, 2, 4, 12, 5] ]
2. # sorting every sublist of the above list
3. sort\_List = **lambda** num : ( sorted(n) **for** n **in** num )
4. # Getting the third largest number of the sublist
5. third\_Largest = **lambda** num, func : [ l[ len(l) - 2] **for** l **in** func(num)]
6. result = third\_Largest( my\_List, sort\_List)
7. **print**( result ) //[6, 54, 5]

# Python File Handling

In Python, files are treated in two modes as text or binary. The file may be in the text or binary format, and each line of a file is ended with the special character.

Hence, a file operation can be done in the following order.

* Open a file
* Read or write - Performing operation
* Close the file

## Opening a file

Python provides an **open()** function that 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.

**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.

| **Access mode** | **Description** |
| --- | --- |
| r | It opens the file to read-only mode. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed. |
| rb | It opens the file to read-only in binary format. The file pointer exists at the beginning of the file. |
| r+ | It opens the file to read and write both. The file pointer exists at the beginning of the file. |
| rb+ | It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file. |
| 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. |
| 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. The file pointer exists at the beginning of the file. |
| 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.** |
| wb+ | It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file. |
| 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. |
| 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. |
| 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. |
| ab+ | It opens a file to append and read both in binary format. The file pointer remains at the end of the file. |

1. fileptr = open("file.txt","r") #opens the file file.txt in read mode
2. **if** fileptr:
3. **print**("file is opened successfully")
4. fileptr.close() #closes the opened file

The **fileptr** holds the file object and if the file is opened successfully, it will execute the print statement.

## The with statement

The with statement is useful in the case of manipulating the files. It is used in the scenario where a pair of statements is to be executed with a block of code in between.

1. with open(<file name>, <access mode>) as <file-pointer>:
2. #statement suite

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always suggestible to use the **with** statement in the case of files because, if break, return, or exception occurs in nested block of code then it automatically closes the file,

we don't need to write **close()** function. It doesn't let the file to corrupt.

1. with open("file.txt",'r') as f:
2. content = f.read();
3. **print**(content)

## 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.

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

**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.

1. # open the file.txt in append mode. Create a new file if no such file exists.
2. fileptr = open("file2.txt", "w")
3. # appending the content to the file
4. fileptr.write('''''Python is the modern day language. It makes things so simple. ''')
5. fileptr.close()

To read a file using the Python script, the Python provides the **read()** method. The **read()** method reads a string from file. It can read data in the text as well as a binary format.

**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.

1. fileptr = open("ab.txt","r") #open file in read mode. causes err if no such file exits.
2. content = fileptr.read(10) #stores all the data of the file into the variable content
3. **print**(type(content)) # prints the type of the data stored in the file
4. **print**(content) #prints the content of the file
5. fileptr.close() #closes the opened file

### Read file through for loop

1. fileptr = open("ab.txt","r"); #open file in read mode.cause an err if no such file exit.
2. **for** i **in** fileptr: #running a for loop
3. **print**(i) # i contains each line of the file

## Read Lines of the file

Python facilitates to read the file line by line by using a function **readline()** method. 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.

Python provides also the **readlines()** method which is used for the reading lines. It returns the list of the lines till the end of **file(EOF)** is reached.

1. fileptr = open("file2.txt","r");
2. content = fileptr.readline() #read one line at a time
3. content1 = fileptr.readline()
4. #read all the lines of a line
5. content = fileptr.readlines() #stores all the data of file into the variable content
6. **print**(content)
7. **print**(content1)
8. fileptr.close()

## File Pointer positions

Python provides the tell() method which is used to print the byte number at which the file pointer currently exists.

1. fileptr = open("file2.txt","r")
2. **print**("The filepointer is at byte :",fileptr.tell()) #initially the filepointer is at 0
3. content = fileptr.read(); #reading the content of the file
4. #after read operation file pointer modifies. tell() returns the location of the fileptr.
5. **print**("After reading, the filepointer is at:",fileptr.tell())

## Modifying file pointer position

In real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

For this purpose, the Python provides us the seek() method which enables us to modify the file pointer position externally.

1. <file-ptr>.seek(offset[, **from**)

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.

1. fileptr = open("file2.txt","r")
2. **print**("The filepointer is at byte :",fileptr.tell()) #initially the filepointer is at 0
3. fileptr.seek(10); #changing the file pointer location to 10.
4. **print**("After reading, filepointer is at:",fileptr.tell()) #tell() returns location of fileptr.

## Python OS module

### Renaming the file: The Python **os** module enables interaction with the operating system. The os module provides functions that are involved in file processing operations like renaming, deleting, etc. It provides us the rename() method to rename the specified file to a new name.

**rename(current-name, new-name)**

1. **import os**
2. **os.rename("file2.txt","file3.txt") #rename file2.txt to file3.txt**

### Removing the file

The os module provides the remove() method which is used to remove the specified file.

**remove(file-name)**

1. **import os;**
2. **os.remove("file3.txt") #deleting the file named file3.txt**

## Creating the new directory

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

**mkdir(directory name)**

## The getcwd() method

This method returns the current working directory.

**os.getcwd()**

## Changing the current working directory

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

**chdir("new-directory")**

## Deleting directory: rmdir() method is used to delete the specified directory.

1. **import os**
2. **os.rmdir("directory\_name") #removing the new directory**

## Writing Python output to the files

In Python, there are the requirements to write the output of a Python script to a file.

The check\_call() method of module subprocess is used to execute a Python script and write the output of that script to a file.

## The file related methods

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

| **Method** | **Description** |
| --- | --- |
| file.close() | It closes the opened file. The file once closed, it can't be read or write anymore. |
| File.fush() | It flushes the internal buffer. |
| File.fileno() | It returns file descriptor used by underlying implementation to request I/O from OS. |
| File.isatty() | It returns true if the file is connected to a TTY device, otherwise returns false. |
| File.next() | It returns the next line from the file. |
| File.read([size]) | It reads the file for the specified size. |
| File.readline([size]) | It reads one line from file and places the file pointer to the beginning of the new line. |
| 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. |
| File.seek(offset[,from) | It modifies position of file pointer to a specified offset with the specified reference. |
| File.tell() | It returns the current position of the file pointer within the file. |
| File.truncate([size]) | It truncates the file to the optional specified size. |
| File.write(str) | It writes the specified string to a file |
| File.writelines(seq) | It writes a sequence of the strings to a file. |

# Python Modules

## What is Modular Programming?

Modular programming is the practice of segmenting a single, complicated coding task into multiple, simpler, easier-to-manage sub-tasks. We call these subtasks modules. Therefore, we can build a bigger program by assembling different modules that act like building blocks.

In Python, modularization of the code is encouraged through the use of functions, modules, and packages.

## What are Modules in Python?

A document with definitions of functions and various statements written in Python is called a Python module.

A module is a file containing Python code, definitions of functions, statements, or classes. An example\_module.py file is a module we will create and whose name is example\_module.

**import example\_module**

1. **result = example\_module.square( 4 )**
2. **print( "By using the module square of number is: ", result )**

**Where example\_module is another file.**

### Importing and also Renaming

While importing a module, we can change its name too.

1. **import math as mt # We will import math module and give a different name to it**
2. **print( "The value of euler's number is", mt.e )**

### Python from...import Statement

We can import specific names from a module without importing the module as a whole.

1. **from math import e, tau # how to import multiple objects from a module**
2. **from math import e #how to import single objects from a module**
3. **print( "The value of euler's number is", e )**

### Import all Names - From import \* Statement

To import all the objects from a module within the present namespace, use the \* symbol and the from and import keyword.

**from** name\_of\_module **import** \*

1. **from** math **import** \* **# importing the complete math module using \***

## The dir() Built-in Function

We may use the dir() method to identify names declared within a module.

1. **print**( "List of functions:\n ", dir( str ), end=", " ) #to print the directory of a module

## Namespaces and Scoping

Objects are represented by names or identifiers called variables. A namespace is a dictionary containing names of variables (keys) & objects that go with them (values).

Both local and global namespace variables can be accessed by a Python statement. When two variables with the same name are local and global, the local variable takes the role of the global variable

1. Number = 204
2. **def** AddNumber():
3. **global** Number # accessing the global namespace
4. Number = Number + 200
5. **print**( Number ) //204
6. AddNumber()
7. **print**( Number ) //404

# Python Exceptions

An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.

### Exceptions versus Syntax Errors

When the interpreter identifies a statement that has an error, syntax errors occur.

## Try and Except Statement - Catching Exceptions

In Python, we catch exceptions and handle them using try and except code blocks. The try clause contains the code that can raise an exception, while the except clause contains the code lines that handle the exception.

1. a = ["Python", "Exceptions", "try and except"]
2. **try**:
3. **for** i **in** range( 4 ):
4. **print**( "The index and element from the array is", i, a[i] )
5. **except**:
6. **print** ("Index out of range")

## How to Raise an Exception

If a condition does not meet our criteria but is correct according to the Python interpreter, we can intentionally raise an exception using the raise keyword. We can use a customized exception in conjunction with the statement.

If we wish to use raise to generate an exception when a given condition happens, we may do so as follows:

1. num = [3, 4, 5, 7] #Python code to show how to raise an exception in Python
2. **if** len(num) > 3:
3. **raise** Exception( f"Length of given list must be less than 3 but it is {len(num)}" )

## Assertions in Python

When we're finished verifying the program, an assertion is a consistency test that we can switch on or off.

The simplest way to understand an assertion is to compare it with an if-then condition. An exception is thrown if the outcome is false when an expression is evaluated.

Assertions are commonly used at the beginning of a function to inspect for valid input and at the end of calling the function to inspect for valid output.

**assert Expressions[, Argument]**

1. **def** square\_root( Number ): # program to show how to use assert keyword
2. **assert** ( Number < 0), "Give a positive integer"
3. **return** Number\*\*(1/2)
4. **print**( square\_root( 36 ) )
5. **print**( square\_root( -36 ) )

## Try with Else Clause

Python also supports the else clause, which should come after every except clause, in the try, and except blocks. Only when the try clause fails to throw an exception the Python interpreter goes on to the else block.

1. **def** reciprocal( num1 ): # Defining a functin which returns reciprocal of a number
2. **try**:
3. reci = 1 / num1
4. **except** ZeroDivisionError:
5. **print**( "We cannot divide by zero" )
6. **else**:
7. **print** ( reci )
8. reciprocal( 4 )
9. reciprocal( 0 )

## Finally Keyword in Python

The finally keyword is available in Python, and it is always used after the try-except block. The finally code block is always executed after the try block has t

1. **try**:
2. div = 4 // 0
3. **print**( div )
4. **except** ZeroDivisionError:
5. **print**( "Atepting to divide by zero" )
6. **finally**: # this will always be executed no matter exception is raised or not
7. **print**( 'This is code of finally clause' )

## Exceptions List: Here is complete list of Python in-built exceptions.

| **Exception Name** | **Description of the Exception** |
| --- | --- |
| **Exception** | All exceptions of Python have a base class. |
| **StopIteration** | If the next() method returns null for an iterator, this exception is raised. |
| **SystemExit** | The sys.exit() procedure raises this value. |
| **StandardError** | Excluding the StopIteration and SystemExit, It is the base class for all Python built-in exceptions. |
| **ArithmeticError** | All mathematical computation errors belong to this base class. |
| **OverflowError** | This exception is raised when a computation surpasses the numeric data type's maximum limit. |
| **FloatingPointError** | If a floating-point operation fails, this exception is raised. |
| **ZeroDivisionError** | For all numeric data types, its value is raised whenever a number is attempted to be divided by 0. |
| **AssertionError** | If the Assert statement fails, this exception is raised. |
| **AttributeError** | This exception is raised if a variable reference or assigning a value fails. |
| **EOFError** | When the endpoint of the file is approached, and the interpreter didn't get any input value by raw\_input() or input() functions, this exception is raised. |
| **ImportError** | This exception is raised if using the import keyword to import a module fails. |
| **KeyboardInterrupt** | If user interrupts the execution of a program, generally by hitting Ctrl+C, this exception is raised. |
| **LookupError** | LookupErrorBase is the base class for all search errors. |
| **IndexError** | This exception is raised when the index attempted to be accessed is not found. |
| **KeyError** | When the given key is not found in the dictionary to be found in, this exception is raised. |
| **NameError** | This exception is raised when a variable isn't located in either local or global namespace. |
| **UnboundLocalError** | This exception is raised when we try to access a local variable inside a function, and the variable has not been assigned any value. |
| **EnvironmentError** | All exceptions that arise beyond the Python environment have this base class. |
| **IOError** | If an input or output action fails, like when using the print command or the open() function to access a file that does not exist, this exception is raised. |
| **SyntaxError** | This exception is raised whenever a syntax error occurs in our program. |
| **IndentationError** | This exception was raised when we made an improper indentation. |
| **SystemExit** | This exception is raised when the sys.exit() method is used to terminate the Python interpreter. The parser exits if the situation is not addressed within the code. |
| **TypeError** | This exception is raised whenever a data type-incompatible action or func is tried to be executed. |
| **ValueError** | This exception is raised if the parameters for a built-in method for a particular data type are of the correct type but have been given the wrong values. |
| **RuntimeError** | This exception is raised when error that occurred during program's execution can’t be classified. |
| **NotImplementedError** | If an abstract function that the user must define in an inherited class is not defined, this exception is raised. |

1. **try**:
2. # Code block
3. # These statements are those which can probably have some error
4. **except**:
5. # This block is optional.
6. # If the try block encounters an exception, this block will handle it.
7. **else**:
8. # If there is no exception, this code block will be executed by the interpreter
9. **finally**:
10. # Python interpreter will always execute this code.

# Python Date and time

In Python, the date is not a data type, but we can work with the date objects by importing the module named with **datetime, time, and calendar**.The **datetime** classes are classified in the six main classes.

* **date** - It is a naive ideal date. It consists of the year, month, and day as attributes.
* **time** - It is a perfect time, assuming every day has precisely 24\*60\*60 seconds. It has hour, minute, second, microsecond, and **tzinfo** as attributes.
* **datetime** - It is a grouping of date and time, along with the attributes year, month, day, hour, minute, second, microsecond, and tzinfo.
* **timedelta -** It represents the difference between two dates, time or datetime instances to microsecond resolution.
* **tzinfo** - It provides time zone information objects.
* **timezone -** It is included in the new version of Python. It is the class that implements the **tzinfo** abstract base class.

## Tick

In Python, the time instants are counted since 12 AM, 1st January 1970. The function **time()** of the module time returns the total number of ticks spent since 12 AM, 1st January 1970. A tick can be seen as the smallest unit to measure the time.

## How to get the current time?

The localtime() functions of the time module are used to get the current time tuple.

1. **import** time;
2. **print**(time.localtime(time.time())) #returns a time tuple

## Time tuple

The time is treated as the tuple of 9 numbers. Let's look at members of the time tuple.

| **Index** | **Attribute** | **Values** |
| --- | --- | --- |
| 0 | Year | 4 digit (for example 2018) |
| 1 | Month | 1 to 12 |
| 2 | Day | 1 to 31 |
| 3 | Hour | 0 to 23 |
| 4 | Minute | 0 to 59 |
| 5 | Second | 0 to 60 |
| 6 | Day of weak | 0 to 6 |
| 7 | Day of year | 1 to 366 |
| 8 | Daylight savings | -1, 0, 1 , or -1 |

## Getting formatted time

The time can be formatted by using the **asctime()** function of the time module. It returns the formatted time for the time tuple being passed.

1. **import** time
2. **print**(time.asctime(time.localtime(time.time()))) #Tue Dec 18 15:31:39 2018

## Python sleep time

## The **sleep()** method of time module is used to stop the execution of script for a given

amount of time. output will be delayed for the number of seconds provided as the float.

1. **import** time
2. **for** i **in** range(0,5):
3. **print**(i)
4. time.sleep(1) #Each element will be printed after 1 second

## The datetime Module

The **datetime** module enables us to create the custom date objects, perform various operations on dates like the comparison, etc.

1. **import** datetime
2. **print**(datetime.datetime.now()) #returns the current datetime object

## Creating date objects

We can create the date objects bypassing the desired date in the datetime constructor for which the date objects are to be created.

1. **import** datetime
2. **print**(datetime.datetime(2020,04,04)) #returns datetime object for specified date

## Comparison of two dates

We can compare two dates by using the comparison operators like >, >=, <, and <=.

1. **from** datetime **import** datetime as dt
2. #Compares the time. If the time is in between 8AM and 4PM, then it prints working hours otherwise it prints fun hours
3. **if** dt(dt.now().year,dt.now().month,dt.now().day,8)<dt.now()<dt(dt.now().year,dt.now().month,dt.now().day,16):
4. **print**("Working hours....")
5. **else**:
6. **print**("fun hours")

## The calendar module

It provides a calendar object that contains various methods to work with the calendars.

1. **import** calendar;
2. cal = calendar.month(2020,3)
3. **print**(cal) #printing the calendar of December 2018

# Python Regex

A regular expression is a set of characters with highly specialized syntax that we can use to find or match other characters or groups of characters. In short, regular expressions, or Regex, are widely used in the UNIX world.

# Python Sending Email using SMTP

SMTP is used as a protocol to handle the email transfer using Python. It is used to route emails between email servers. It is an application layer protocol which allows to users to send mail to another. The receiver retrieves email using the protocols **POP(Post Office Protocol)** and **IMAP(Internet Message Access Protocol)**.

Python provides a **smtplib** module, which defines an the SMTP client session object used to send emails to an internet machine.

1. **import** smtplib
2. smtpObj = smtplib.SMTP(host, port, local\_hostname)

It accepts the following parameters.

* **host:** It is the hostname of the machine which is running your SMTP server. Here, we can specify the IP address of the server like ([https://www.abc.com](https://www.abc.com/)) or localhost. It is an optional parameter.
* **port:** It is the port number on which the host machine is listening to the SMTP connections. It is 25 by default.
* **local\_hostname:** If the SMTP server is running on your local machine, we can mention the hostname of the local machine.

The sendmail() method of the SMTP object is used to send the mail to the desired machine.

**smtpObj.sendmail(sender, receiver, message)**

1. **import** smtplib
2. sender\_mail = 'sender@fromdomain.com'
3. receivers\_mail = ['reciever@todomain.com']
4. message = """From: From Person %s """%(sender\_mail,receivers\_mail)
5. **try**:
6. smtpObj = smtplib.SMTP('localhost')
7. smtpObj.sendmail(sender\_mail, receivers\_mail, message)
8. **print**("Successfully sent email")
9. **except** Exception:
10. **print**("Error: unable to send email")

## Sending email from gmail

There are cases where the emails are sent using the Gmail SMTP server. In this case, we can pass Gmail as the SMTP server instead of using the localhost with the port 587.

smtpObj = smtplib.SMTP("gmail.com", 587)

Here, we need to login to the Gmail account using Gmail user name and password. For this purpose, the smtplib provide the login() method, which accepts the username and password of the sender.

This may make your Gmail ask you for access to less secure apps if you're using Gmail. You will need to turn this ON temporarily for this to work.

1. **import** smtplib
2. sender\_mail = 'sender@gmail.com'
3. receivers\_mail = ['reciever@gmail.com']
4. message = """From: From Person e. """%(sender\_mail,receivers\_mail)
5. **try**:
6. password = input('Enter the password');
7. smtpObj = smtplib.SMTP('gmail.com',587)
8. smtpobj.login(sender\_mail,password)
9. smtpObj.sendmail(sender\_mail, receivers\_mail, message)
10. **print**("Successfully sent email")
11. **except** Exception:
12. **print**("Error: unable to send email")

## Sending HTML in email

We can format the HTML in the message by specifying the MIME version, content-type, and character set to send the HTML.

1. **import** smtplib
2. sender\_mail = 'sender@fromdomain.com'
3. receivers\_mail = ['reciever@todomain.com']
4. message = """From: From Person %s
5. To: To Person %s
6. MIME-Version:1.0
7. Content-type:text/html
8. Subject: Sending SMTP e-mail
9. <h3>Python SMTP</h3>
10. <strong>This is a test e-mail message.</strong>
11. """%(sender\_mail,receivers\_mail)
12. **try**:
13. smtpObj = smtplib.SMTP('localhost')
14. smtpObj.sendmail(sender\_mail, receivers\_mail, message)
15. **print**("Successfully sent email")
16. **except** Exception:
17. **print**("Error: unable to send email")

# Python read csv file

## Python CSV Module Functions

The CSV module work is used to handle the CSV files to read/write and get data from specified columns. There are different types of CSV functions, which are as follows:

* **csv.field\_size\_limit -** It returns current maximum field size allowed by the parser.
* **csv.get\_dialect -** It returns the dialect associated with a name.
* **csv.list\_dialects -** It returns the names of all registered dialects.
* **csv.reader -** It read the data from a csv file
* **csv.register\_dialect -** It associates dialect with a name. The name must be a string or a Unicode object.
* **csv.writer -** It writes the data to a csv file
* **o csv.unregister\_dialect -** It deletes the dialect which is associated with the name from the dialect registry. If a name is not a registered dialect name, then an error is being raised.
* **csv.QUOTE\_ALL -** It instructs the writer objects to quote all fields. csv.QUOTE\_MINIMAL - It instructs the writer objects to quote only those fields which contain special characters such as quotechar, delimiter, etc.
* **csv.QUOTE\_NONNUMERIC -** It instructs the writer objects to quote all the non-numeric fields.
* **csv.QUOTE\_NONE -** It instructs the writer object never to quote the fields.

## Reading CSV files

Python provides various functions to read csv file. here are a few method of reading csv.

* **Using csv.reader() function**

In Python, the **csv.reader()** module is used to read the csv file. It takes each row of the file and makes a list of all the columns.

1. **import** csv
2. with open('python.csv') as csv\_file:
3. csv\_reader = csv.reader(csv\_file, delimiter=',')
4. line\_count = 0
5. **for** row **in** csv\_reader:
6. **if** line\_count == 0:
7. **print**(f'Column names are {", ".join(row)}')
8. line\_count += 1

## Read a CSV into a Dictionar

We can also use **DictReader()** function to read the csv file directly into a dictionary rather than deal with a list of individual string elements.

1. **import** csv
2. with open('python.txt', mode='r') as csv\_file:
3. csv\_reader = csv.DictReader(csv\_file)
4. line\_count = 0
5. **for** row **in** csv\_reader:
6. **if** line\_count == 0:
7. **print**(f'The Column names are as follows {", ".join(row)}')
8. line\_count += 1
9. **print**(f'\t{row["name"]} works in the {row["department"]} department, and was born in {row["birthday month"]}.')
10. line\_count += 1
11. **print**(f'Processed {line\_count} lines.')

## Reading csv files with Pandas

The Pandas is defined as an open-source library which is built on the top of the NumPy library. It provides fast analysis, data cleaning, and preparation of the data for the user.

Reading the csv file into a pandas **DataFrame** is quick and straight forward. We don't need to write enough lines of code to open, analyze, and read the csv file in pandas and it stores the data in **DataFrame**.

1. **import** pandas
2. df = pandas.read\_csv('hrdata.csv')
3. **print**(df)

In the above code, the three lines are enough to read the file, and only one of them is doing the actual work, i.e., pandas.read\_csv()

# Python Write CSV File

We can also write any new and existing CSV files in Python by using the csv.writer() module. It is similar to the csv.reader() module and also has two methods, i.e., **writer** function or the **Dict Writer** class.

It presents two functions, i.e., **writerow()** and **writerows()**. The **writerow()** function only write one row, and the **writerows()** function write more than one row.

**Dialects:-**It is defined as a construct that allows you to create, store, and re-use various formatting parameters. It supports several attributes; the most frequently used are:

* **Dialect.delimiter:** This attribute is used as the separating character between the fields. The default value is a comma (,).
* **Dialect.quotechar:** This attribute is used to quote fields that contain special char.
* **Dialect.lineterminator:** It is used to create new lines, and default value is '\r\n'.

1. **import** csv
2. with open('Python.csv', 'w') as csvfile:
3. fieldnames = ['first\_name', 'last\_name', 'Rank']
4. writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
5. writer.writeheader()
6. writer.writerow({'Rank': 'B', 'first\_name': 'Jane', 'last\_name': 'Loive'})
7. print("Writing complete")

## Write a CSV into a Dictionary

We can also use the class **DictWriter** to write the CSV file directly into a dictionary.

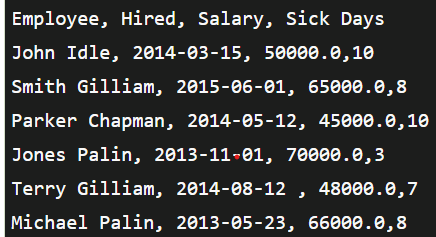
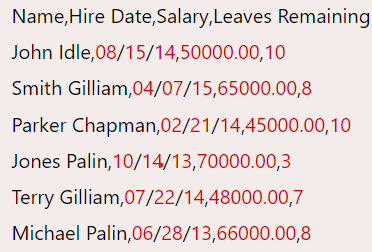
1. **import** csv
2. with open('python.csv', mode='w') as csv\_file:
3. fieldnames = ['emp\_name', 'dept', 'birth\_month']
4. writer = csv.DictWriter(csv\_file, fieldnames=fieldnames)
5. writer.writeheader()
6. writer.writerow({'emp\_name': 'Smith', 'dept': 'IT', 'birth\_month': 'October'})

## Writing CSV Files Using Pandas

It is as easy as reading the CSV file using pandas. You need to create the DataFrame, which is a two-dimensional, heterogeneous tabular data structure and consists of three main components- data, columns, and rows.

1. **import** pandas
2. df = pandas.read\_csv('hrdata.csv',
3. index\_col='Employee',
4. parse\_dates=['Hired'],
5. header=0,
6. names=['Employee', 'Hired', 'Salary', 'Sick Days'])
7. df.to\_csv('hrdata\_modified.csv')

I/P O/P



# Python read excel file

## Excel Documents

The first row of the spreadsheet is mainly reserved for the header, while the first column identifies the sampling unit. Each workbook can contain multiple sheets that are also called a worksheets. A box at a particular column and row is called a cell, and each cell can include a number or text value. The grid of cells with data forms a sheet.

The active sheet is defined as a sheet in which the user is currently viewing or last viewed before closing Excel.

## Reading from an Excel file

First, you need to write a command to install the **xlrd** module.

pip install xlrd

1. **import** xlrd
2. loc = ("path of file") # Define the location of the file
3. wb = xlrd.open\_workbook(loc) # To open the Workbook
4. sheet = wb.sheet\_by\_index(0)
5. sheet.cell\_value(0, 0) # For row 0 and column 0

## Reading from the Pandas

It is a python package which provides a beneficial data structure called a data frame.

1. **import** pandas as pd
2. data = pd.read\_csv(".csv", low\_memory=False) # Read the file
3. **print**("Total rows: {0}".format(len(data))) # Output the number of rows
4. **print**(list(data)) # See which headers are available

## Reading from the openpyxl

First, we need to install an openpyxl module using pip from the command line.

pip install openpyxl

We can also read data from the existing spreadsheet using openpyxl. It also allows the user to perform calculations and add content that was not part of the original dataset.

1. **import** openpyxl
2. my\_wb = openpyxl.Workbook()
3. my\_sheet = my\_wb.active
4. my\_sheet\_title = my\_sheet.title
5. **print**("My sheet title: " + my\_sheet\_title)

# Python Write Excel File

The Python write excel file is used to perform the multiple operations on a spreadsheet using the **xlwt** module. It is an ideal way to write data and format information to files with .xls extension.

If you want to write data to any file and don't want to go through the trouble of doing everything by yourself, then you can use a for loop to automate the whole process a little bit.

## Write Excel File Using xlsxwriter Module

We need to use the following command to install xlsxwriter module:

pip install xlsxwriter

## Write Excel File Using openpyxl Module

It is defined as a package which is generally recommended if you want to read and write .xlsx, xlsm, xltx, and xltm files. You can check it by running **type(wb)**.

The load\_workbook() function takes an argument and returns a workbook object, which represents the file. Make sure that you are in the same directory where your spreadsheet is located. Otherwise, you will get an error while importing.

You can easily use a for loop with the help of the range() function to help you to print out the values of the rows that have values in column 2. If those particular cells are empty, you will get None.

## Writing Files with pyexcel

You can easily export your arrays back to a spreadsheet by using the save\_as() function and pass the array and name of the destination file to the dest\_file\_name argument.

It allows us to specify the delimiter and add dest\_delimiter argument. You can pass the symbol that you want to use as a delimiter in-between " ".

1. **import** xlsxwriter
2. book = xlsxwriter.Book('Example2.xlsx')
3. sheet = book.add\_sheet()
4. row = 0 # Rows and columns are zero indexed.
5. column = 0
6. content = ["Parker", "Smith", "John"]
7. **for** item in content : # iterating through the content list
8. sheet.write(row, column, item) # write operation perform
9. row += 1 # incrementing the value of row by one with each iterations.
10. book.close()

# Python Assert Keyword

Python assert keyword is defined as a debugging tool that tests a condition. The Assertions are mainly the assumption that asserts or state a fact confidently in the program. For example, while writing a division function, the divisor should not be zero, and you assert that the divisor is not equal to zero.

**assert** condition, error\_message(optional)

## Why Assertion is used

It is a debugging tool, and its primary task is to check the condition. If it finds that the condition is true, it moves to the next line of code, and If not, then stops all its operations and throws an error. It points out the error in the code.

## Where Assertion in Python used

* Checking the outputs of the functions.
* Used for testing the code.
* In checking the values of arguments.Checking the valid input.

1. **def** avg(scores):
2. **assert** len(scores) != 0,"The List is empty."
3. **return** sum(scores)/len(scores)
4. scores2 = [67,59,86,75,92]
5. **print**("The Average of scores2:",avg(scores2))
6. scores1 = []
7. **print**("The Average of scores1:",avg(scores1)) //The list is empty.

# Python List Comprehension

Python is known for helping us produce code that is elegant, simple to write, and reads almost as well as plain English. List comprehension is one of the language's most distinguishing features, allowing us to develop sophisticated functionality with just one line of code.

newlist = [expression **for** item **in** iterable **if** condition == True]

1. numbers = [3, 5, 1, 7, 3, 9] #using list comprehension to iterate through list items
2. num = [n\*\*2 **for** n **in** numbers]
3. **print**(num) //[9, 25, 1, 49, 9, 81]

List comprehensions are easier to read and grasp than loops since they are more declarative.

# Python Collection Module

The Python collection module is defined as a container that is used to store collections of data, for example - list, dict, set, and tuple, etc. It was introduced to improve the functionalities of the built-in collection containers.

There are different types of collection modules which are as follows:

## namedtuple()

The Python **namedtuple()** function returns a tuple-like object with names for each position in the tuple. It was used to eliminate the problem of remembering the index of each field of a tuple object in ordinary tuples.

1. pranshu = ('James', 24, 'M')
2. **print**(pranshu) //('James', 24, 'M')

## OrderedDict()

The Python OrderedDict() is similar to a dictionary object where keys maintain the order of insertion. If we try to insert key again, previous value will be overwritten for that key.

1. **import** collections
2. d1=collections.OrderedDict()
3. d1['A']=10
4. d1['C']=12
5. **for** k,v **in** d1.items():
6. **print** (k,v)

## defaultdict()

The Python defaultdict() is defined as a dictionary-like object. It is a subclass of the built-in dict class. It provides all methods provided by dictionary but takes the first argument as a default data type.

1. **from** collections **import** defaultdict
2. number = defaultdict(int)
3. number['one'] = 1
4. number['two'] = 2
5. **print**(number['three']) //0

## Counter()

Python Counter is subclass of dictionary object which helps to count hashable objects.

1. **from** collections **import** Counter
2. c = Counter()
3. list = [1,2,3,4,5,7,8,5,9,6,10]
4. Counter(list)
5. Counter({1:5,2:4})
6. list = [1,2,4,7,5,1,6,7,6,9,1]
7. c = Counter(list)
8. **print**(c[1]) //3

## deque()

The Python **deque()** is a double-ended queue which allows us to add and remove elements from both the ends.

1. **from** collections **import** deque
2. list = ["x","y","z"]
3. deq = deque(list)
4. **print**(deq) //deque(['x', 'y', 'z'])

## Chainmap Objects

A **chainmap** class is used to groups multiple dictionary together to create a single list. The linked dictionary stores in list and it is public and can be accessed by map attribute.

1. **from** collections **import** ChainMap
2. baseline = {'Name': 'Peter', 'Age': '14'}
3. adjustments = {'Age': '14', 'Roll\_no': '0012'}
4. **print**(list(ChainMap(adjustments, baseline))) //['Name', 'Age', 'Roll\_no' ]

## UserDict Objects

The UserDict behaves as a wrapper around the dictionary objects. The dictionary can be accessed as an attribute by using the **UserDict** object. It provides the easiness to work with the dictionary.

It provides the following attribute.

**data** - A real dictionary used to store the contents of the UserDict class.

## 

## UserList Objects

The UserList behaves as a wrapper class around the list-objects. It is useful when we want to add new functionality to the lists. It provides the easiness to work with the dictionary.

It provides the following attribute.

**data** - A real list is used to store the contents of the User class.

## 

## UserString Objects

The **UserList** behaves as a wrapper class around the list objects. The dictionary can be accessed as an attribute by using the **UserString** object. It provides the easiness to work with the dictionary.

It provides the following attribute.

**data** - A real **str** object is used to store the contents of the UserString class.

# Python Math Module

### Calculating the Ceiling and the Floor Value

1. **import** math
2. x = 4.346
3. **print**( math.ceil(x) ) //5
4. **print**( math.floor(x) ) //4

### Calculating the Factorial of the Number

**print**( "The factorial of 6 is : ", math.factorial(6) )

### Calculating the Absolute Value

The method math.fabs() returns the absolute number of the number given to function.

1. **import** math
2. x = -45
3. **print**( "The absolute value of -45 is: ", math.fabs(x) ) //45.0

### Calculating the Exponential

x to the power of e, often known as the exponential of a number x, is calculated using the exp() function.

1. **import** math
2. num1 = 4
3. num2 = -3
4. num3 = 0.00
5. **print**( f" exponenetial val of {num1} is: ", math.exp(num1) ) 54.598150033144236
6. **print**( f" expo value of {num2} is: ", math.exp(num2) ) //0.049787068367863944
7. **print**( f" exponenetial value of {num3} is: ", math.exp(num3) ) //1.0

### Calculating the Power of a Number

x\*\*y is computed via the pow() function. This function calculates the value of the power after converting its inputs into floats.

1. **import** math
2. x = 4
3. y = 5
4. **print**( f"The value of {x} to the power of {y} is: ", math.pow(x,y) ) //1024.0

### Calculating Sine, Cosine, and Tangent

The values of sine, cosine, and tangent of an angle, which are supplied as an input to the function, are returned by the sin(), cos(), and tan() methods. This function expects a value that is provided in radians.

1. **import** math
2. angle = math.pi / 4
3. **print**( "The sine of pi/4 is : ", math.sin( angle ) ) //0.7071067811865475

## Description of all the Functions in Python Math Module

| **Function** | **Description** |
| --- | --- |
| **ceil(x)** | The lowest integer bigger than or equal to x is returned. |
| **copysign(x, y)** | gives x back with the sign of y. |
| **fabs(x)** | gives x's absolute value back. |
| **factorial(x)** | provides the x factorial back. |
| **floor(x)** | gives back the biggest integer that is less than or equal to x. |
| **fmod(x, y)** | returns the leftover value after dividing x by y. |
| **frexp(x)** | returns the pair of the mantissa and exponent of x. (m, e) |
| **fsum(iterable)** | returns the iterable's correct floating point sum of all values. |
| **isfinite(x)** | If x is neither an infinity nor a NaN, it returns True (Not a Number) |
| **isinf(x)** | If x is a positive or negative infinity, it returns True. |
| **isnan(x)** | If x is a NaN, it returns True. |
| **ldexp(x, i)** | gives back x \* (2\*\*i). |
| **modf(x)** | gives x's fractional and integer components back. |
| **trunc(x)** | x's shortened integer value is returned. |
| **exp(x)** | delivers e\*\*x |
| **expm1(x)** | yields e\*\*x - 1 |
| **log(x[, b])** | gives back the x logarithm in base b. (defaults to e) |
| **log1p(x)** | the natural logarithm of 1 + x is returned. |
| **log2(x)** | gives x's base-2 logarithm back. |
| **log10(x)** | provides x's base-10 logarithm. |
| **pow(x, y)** | gives x raised to the power of y back. |
| **sqrt(x)** | gives x's square root back. |
| **acos(x)** | gives the arc cosine of x back. |
| **asin(x)** | gives the arc sine of x back. |
| **atan(x)** | gives the arc tangent of x back. |
| **atan2(y, x)** | gives back atan(y / x). |
| **cos(x)** | returns the x's cosine. |
| **hypot(x, y)** | returns sqrt(x\*x + y\*y), the Euclidean norm. |
| **sin(x)** | gives the sine of x back. |
| **tan(x)** | gives the tangent of x back. |
| **degrees(x)** | Angle x is transformed from radians to degrees. |
| **radians(x)** | Angle x is transformed from degrees to radians. |
| **acosh(x)** | x's inverse hyperbolic cosine is returned. |
| **asinh(x)** | x's inverse hyperbolic sine is returned. |
| **atanh(x)** | x's inverse hyperbolic tangent is returned. |
| **cosh(x)** | gives x's hyperbolic cosine. |
| **sinh(x)** | gives x's hyperbolic cosine. |
| **tanh(x)** | gives x's hyperbolic tangent back. |
| **erf(x)** | the error function at x is returned. |
| **erfc(x)** | a function that gives the complementary error at x |
| **gamma(x)** | the Gamma function at x is returned. |
| **lgamma(x)** | gives the natural logarithm of the gamma function's absolute value at x. |
| **pi** | The ratio of a circle's circumference to its diameter is a mathematical constant (3.14159...) |
| **e** | e is a constant in mathematics (2.71828...) |

# Python OS Module

Python OS module provides the facility to establish the interaction between the user and

the operating system. It offers many useful OS functions that are used to perform OS-based tasks and get related information about operating system.

## os.name()

This function provides the name of the operating system module that it imports.

## os.mkdir()

The **os.mkdir()** function is used to create new directory.

## os.getcwd()

It returns the current working directory(CWD) of the file.

## os.chdir()

The **os** module provides the **chdir()** function to change the current working directory.

1. **import** os
2. os.chdir("d:\\")

## os.rmdir()

The **rmdir()** function removes the specified directory with an absolute or related path. First, we have to change the current working directory and remove the folder.

1. **import** os
2. # It will throw a Permission error; that's why we have to change the current working directory.
3. os.rmdir("d:\\newdir")
4. os.chdir("..")
5. os.rmdir("newdir")

## os.error()

The os.error() function defines the OS level errors. It raises OSError in case of invalid or inaccessible file names and path etc.

## os.popen()

This function opens a file or from the command specified, and it returns a file object which is connected to a pipe.popen() is similar to open()

## os.close()

This function closes the associated file with descriptor **fr**.

## os.rename()

A file or directory can be renamed by using the function **os.rename()**. A user can rename the file if it has privilege to change the file.

## os.access()

This function uses real **uid/gid** to test if the invoking user has access to the path.

1. **import** os
2. **import** sys
3. path1 = os.access("Python.txt", os.F\_OK)
4. **print**("Exist path:", path1)

# Python Random module

The Python Random module is a built-in module for generating random integers in Python. These are sort of fake random numbers which do not possess true randomness. We can therefore use this module to generate random numbers, display a random item for a list or string, and so on.

## Generate Random Floats

The random.random() function gives a float number that ranges from 0.0 to 1.0. There are no parameters required for this function.

**random.random():-** Returns The second random floating point value within [0.0 and 1) is returned.

**random.uniform(a, b):-** Generates a random floating point R in which a <= R <= b if a <= b and b <= R <= a if b < a.

**random.expovariate(lambda):-**Return random valu acording to exponential distribution.

**random.gauss(mu, sigma):-** Returns random value according to gaussian distribution.

There are other distributions also, such as Gamma Distribution, Normal Distribution, etc.

## Generate Random Integers

random.randint() function generates a random integer from range of numbers supplied.

1. **import** random
2. num = random.randint(1, 500)
3. **print**( num ) //215

## Generate Random Numbers within a Defined Range

The random.randrange() function selects an item randomly from the given range defined by the start, the stop, and the step parameters. By default, the start is set to 0. Likewise, the step is set to 1 by default.

1. **import** random
2. num = random.randrange(1, 10, 2)
3. **print**( num ) //9

## Select Random Elements

The random.choice() function selects an item from a non-empty series at random. An IndexError is thrown when the parameter is an empty series.

1. **import** random
2. random\_l = random.choice([23, 54, 765, 23, 45, 45]) #a list
3. **print**( random\_l ) //765

## Shuffle Elements Randomly

A general sequence, like integers or floating-point series, can be a group of things like a List / Set. The random module contains methods that we can use to add randomization to the series.

The random.shuffle() function shuffles the entries in a list at random.

1. a\_list = [34, 23, 65, 86, 23, 43]
2. random.shuffle( a\_list )
3. **print**( a\_list ) //[23, 43, 86, 65, 34, 23]
4. random.shuffle( a\_list )
5. **print**( a\_list ) //[65, 23, 86, 23, 34, 43]

## Random Seed

We normally use the time of the system to ensure that the software delivers a different output each time we execute it because pseudorandom synthesis is dependent on the preceding number. As a result, we employ seeds.

We can specify a seed to have an initial number using Python's random.seed() function. This seed number determines a random number generator's outcome; therefore, if it stays the same, the outcome will continue to be the same.

1. **import** random
2. random.seed(2)
3. **print**('Generating 5 random numbers: ')
4. **print**([ random.randint(1, 300) **for** r **in** range(6)]) //[29, 47, 44, 185, 87, 158]
5. random.seed(2) # Reseting the seed value to 1
6. # We will get the same numbers as before
7. **print**([random.randint(1, 300) **for** i **in** range(6)]) //[29, 47, 44, 185, 87, 158]

## Various Functions of Random Module

Following is the list of functions available in the random module.

| **Function** | **Description** |
| --- | --- |
| **seed(a=None, version=2)** | This function creates a new random number. |
| **getstate()** | This method provides an object reflecting the generator's present state. Provide the argument to setstate() to recover the state. |
| **setstate(state)** | Providing the state object resets the function's state at the time getstate() was invoked. |
| **getrandbits(k)** | This function provides a Python integer having k random bits. This is important for random number production algorithms like randrange(), which can manage arbitrarily huge ranges. |
| **randrange(start, stop[, step])** | From the range, it produces a random integer. |
| **randint(a, b)** | Provides an integer within a and b at random (both inclusive). If a > b, a ValueError is thrown. |
| **choice(seq)** | Produce a non-empty series item at random. |
| **shuffle(seq)** | Change the order. |
| **sample(population, k)** | Display a list of k-size unique entries from the population series. |
| **random()** | This function creates a new random number. |
| **uniform(a, b)** | This method provides an object reflecting the generator's present state. Provide the argument to setstate() to recover the state. |
| **triangular(low, high, mode)** | Providing the state object resets the function's state at the time getstate() was invoked. |
| **betavariate(alpha, beta)** | Beta distribution |
| **expovariate(lambd)** | Exponential distribution |
| **gammavariate(alpha, beta)** | Gamma distribution |
| **gauss(mu, sigma)** | Gaussian distribution |
| **lognormvariate(mu, sigma)** | Log normal distribution |
| **normalvariate(mu, sigma)** | Normal distribution |
| **vonmisesvariate(mu, kappa)** | Vonmises distribution |
| **paretovariate(alpha)** | Pareto distribution |
| **weibullvariate(alpha, beta)** | Weibull distribution |

# Python statistics module

Python statistics module provides the functions to mathematical statistics of numeric data. There are some popular statistical functions defined in this module.

## mean() function

The mean() function is used to calculate the arithmetic mean of the numbers in the list.

1. **import** statistics
2. datasets = [5, 2, 7, 4, 2, 6, 8]
3. x = statistics.mean(datasets)
4. **print**("Mean is :", x) //4.857142857142857

## median() function

The median() function is used to return the middle value of the numeric data in the list.

1. **import** statistics
2. datasets = [4, -5, 6, 6, 9, 4, 5, -2]
3. **print**("Median of data-set is : % s " % (statistics.median(datasets))) //4.5

## mode() function

The mode() function returns the most common data that occurs in the list.

1. **import** statistics
2. dataset =[2, 4, 7, 7, 2, 2, 3, 6, 6, 8]
3. **print**("Calculated Mode % s" % (statistics.mode(dataset))) //2

## stdev() function

The stdev() function is used to calculate the standard deviation on a given sample which is available in the form of the list.

1. **import** statistics
2. sample = [7, 8, 9, 10, 11]
3. **print**("Standard Deviation is % s " % (statistics.stdev(sample))) //1.5811388300

## median\_low()

The median\_low function is used to return the low median of numeric data in the list.

1. **import** statistics
2. set1 = [4, 6, 2, 5, 7, 7]
3. # Note: low median will always be a member of the data-set.
4. **print**("Low median of data-set is % s " % (statistics.median\_low(set1))) //5

## median\_high()

The median\_high function is used to return the high median of numeric data in the list.

1. **import** statistics
2. dataset = [2, 1, 7, 6, 1, 9]
3. **print**("High median of data-set is %s " % (statistics.median\_high(dataset))) //6

# Python sys module

The python sys module provides functions and variables which are used to manipulate different parts of the Python Runtime Environment. It lets us access system-specific parameters and functions.

**import sys**

First, we have to import the sys module in our program before running any functions.

**sys.modules**

This function provides name of the existing python modules which have been imported.

**sys.argv**

This function returns a list of command line arguments passed to a Python script. The name of the script is always the item at index 0, and the rest of the arguments are stored at subsequent indices.

**sys.base\_exec\_prefix**

This function provides an efficient way to the same value as exec\_prefix. If not running a virtual environment, the value will remain the same.

**sys.base\_prefix**

It is set up during Python startup, before site.py is run, to the same value as prefix.

**sys.byteorder**

It is an indication of the native byteorder that provides an efficient way to do something.

**Sys.maxsize:** This function returns the largest integer of a variable.

**Sys.path :** This function shows the PYTHONPATH set in the current system. It is an environment variable that is a search path for all the python modules.

**sys.stdin**

It is an object that contains the original values of stdin at the start of the program and used during finalization. It can restore the files.

**Sys.getrefcount :** This function returns the reference count of an object.

**sys.exit:**This function is used to exit from either the Python console or command prompt, and also used to exit from the program in case of an exception.

**sys executable**

The value of this function is the absolute path to a Python interpreter. It is useful for knowing where python is installed on someone else machine.

**sys.platform**

This value of this function is used to identify the platform on which we are working.