1. **Function**

**Defining a Function:**

1. Function blocks begin with the keyword def followed by the function name and

Parentheses (( )).

1. The first statement of a function can be an optional statement - the documentation

String of the function or docstring.

1. The code block within every function starts with a colon (:) and is indented.
2. The statement return [expression] exits a function, optionally passing back an

Expression to the caller. A return statement with no arguments is the same as

return None.

def functionname( parameters ):

"function\_docstring"

function\_suite

return [expression]

**Calling a Function:**

Function can be called from another function or from prompt. Once the function is defined we can execute the function by calling it.

def printme(str):

print(str)

return

#Now we can call the function

printme(“Hello”)

printme(“Again Second Hello”)

**Pass by Reference vs Value:**

By default all parameter in python are passed by reference. Means if we change what a parameter refer inside the function then the change also reflect back outside of the function too.

def changeme( mylist):

print (“Values inside the function: ” , mylist)

mylist[2] = 50

print(“values after the change:” , mylist)

return

#now call the changeme function

mylist = [10,20,30]

changeme( mylist)

print(“Values outside the function: ", mylist)

**Output:**

Values inside the function before change: [10, 20, 30]

Values inside the function after change: [10, 20, 50]

Values outside the function: [10, 20, 50]

In Some cases the reference is overridden in called function.

def changeme( mylist ):

"This changes a passed list into this function"

mylist = [1,2,3,4] # This would assi new reference in mylist

print ("Values inside the function: ", mylist)

return

# Now you can call changeme function

mylist = [10,20,30]

changeme( mylist )

print ("Values outside the function: ", mylist)

**Output:**

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

Here the highlighted mylist parameter is local to called function. So changing within the function would not affect outer mylist .

**Passing Parameter to Function:**

Required Argument:

-----------------------

Here we pass the arguments in correct positional order. The number of argument in the function call should match with the number of argument in function definition.

#!/usr/bin/python3

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

print (str)

return

# Now you can call printme function

printme()

It will give error as below:

Traceback (most recent call last):

File "test.py", line 11, in <module>

printme()

TypeError: printme() missing 1 required positional argument: 'str'

KeyWord Argument:

----------------------

This allows you to skip arguments or place them out of order because the Python

interpreter is able to use the keywords provided to match the values with parameters.

#!/usr/bin/python3

# Function definition is here

def printinfo( name, age ):

"This prints a passed info into this function"

print ("Name: ", name)

print ("Age ", age)

return

# Now you can call printinfo function

printinfo( age=50, name="miki" )

When the above code is executed, it produces the following result-

Name: miki

Age 50

Default Argument:

--------------------

A default argument is an argument that assumes a default value if a value is not provided

in the function call for that argument.

#!/usr/bin/python3

# Function definition is here

def printinfo( name, age = 35 ):

"This prints a passed info into this function"

print ("Name: ", name)

print ("Age ", age)

return

# Now you can call printinfo function

printinfo( age=50, name="miki" )

printinfo( name="miki" )

When the above code is executed, it produces the following result-

Name: miki

Age 50

Name: miki

Age 35

Variable-length Arguments

------------------------------

You may need to process a function for more arguments than you specified while defining

the function. These arguments are called *variable-length* arguments and are not named in

the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is given below –

def functionname([formal\_args,] \*var\_args\_tuple ):

"function\_docstring"

function\_suite

return [expression]

An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword

variable arguments. This tuple remains empty if no additional arguments are specified

during the function call. Following is a simple example-

#!/usr/bin/python3

# Function definition is here

def printinfo( arg1, \*vartuple ):

"This prints a variable passed arguments"

print ("Output is: ")

print (arg1)

for var in vartuple:

print (var)

return

# Now you can call printinfo function

printinfo( 10 )

printinfo( 70, 60, 50 )

When the above code is executed, it produces the following result-

Output is:

10

Output is:

70

60

50

**The Anonymous Functions**

This is also called Lambda Function.

Lambda forms can take any number of arguments but return just one value in the

form of an expression. They cannot contain commands or multiple expressions.

An anonymous function cannot be a direct call to print because lambda requires an

expression.

Lambda functions have their own local namespace and cannot access variables

other than those in their parameter list and those in the global namespace.

Syntax:

---------

lambda [arg1 [,arg2,.....argn]]:expression

Following is an example to show how lambda form of function works-

#!/usr/bin/python3

# Function definition is here

sum = lambda arg1, arg2: arg1 + arg2

# Now you can call sum as a function

print ("Value of total : ", sum( 10, 20 ))

print ("Value of total : ", sum( 20, 20 ))

**The return Statement:**

The statement return [expression] exits a function, optionally passing back an expression

to the caller. A return statement with no arguments is the same as return None.

#!/usr/bin/python3

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2

print ("Inside the function : ", total)

return total

# Now you can call sum function

total = sum( 10, 20 )

print ("Outside the function : ", total )

**Scope of Variables**

All variables in a program may not be accessible at all locations in that program. This

depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a

particular identifier. There are two basic scopes of variables in Python-

Global variables

Local variables

**Global vs. Local variables:**

Variables that are defined inside a function body have a local scope, and those defined

outside have a global scope.

This means that local variables can be accessed only inside the function in which they are

declared, whereas global variables can be accessed throughout the program body by all

functions. When you call a function, the variables declared inside it are brought into scope.

#!/usr/bin/python3

total = 0 # This is global variable.

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2; # Here total is local variable.

print ("Inside the function local total : ", total)

return total

# Now you can call sum function

sum( 10, 20 )

print ("Outside the function global total : ", total )

When the above code is executed, it produces the following result-

Inside the function local total : 30

Outside the function global total : 0

1. **Modules:**

Simply, a module is a file consisting of Python code. A module can define functions, classes

and variables. A module can also include runnable code.

Here is an example of a simple module, support.py-

def print\_func( par ):

print "Hello : ", par

return

You can use any Python source file as a module by executing an import statement in some

other Python source file. The import has the following syntax

import module1[, module2[,... moduleN]

When the interpreter encounters an import statement, it imports the module if the module

is present in the search path. A search path is a list of directories that the interpreter

searches before importing a module. For example, to import the module hello.py, you

need to put the following command at the top of the script-

#!/usr/bin/python3

# Import module support

import support

# Now you can call defined function that module as follows

support.print\_func("Zara")

A module is loaded only once, regardless of the number of times it is imported. This

prevents the module execution from happening repeatedly, if multiple imports occur.

**The from...import Statement**

Python's **from** statement lets you import specific attributes from a module into the current

namespace. The **from...import** has the following syntax

from modname import name1[, name2[, ... nameN]]

For example, to import the function fibonacci from the module fib, use the following

statement-

#!/usr/bin/python3

# Fibonacci numbers module

def fib(n): # return Fibonacci series up to n

result = []

a, b = 0, 1

while b < n:

result.append(b)

a, b = b, a+b

return result

>>> **from fib import fib**

>>> fib(100)

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

**The from...import \* Statement:**

It is also possible to import all the names from a module into the current namespace by

using the following import statement-

from modname import \*

**Executing Modules as Scripts:**

Within a module, the module’s name (as a string) is available as the value of the global

variable **\_\_name\_\_.** The code in the module will be executed, just as if you imported it,

but with the \_\_name\_\_ set to "\_\_main\_\_".

#!/usr/bin/python3

# Fibonacci numbers module

def fib(n): # return Fibonacci series up to n

result = []

a, b = 0, 1

while b < n:

result.append(b)

a, b = b, a+b

return result

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

f=fib(100)

print(f)

**Locating Modules:**

When you import a module, the Python interpreter searches for the module in the following

sequences-

The current directory.

If the module is not found, Python then searches each directory in the shell variable

PYTHONPATH.

If all else fails, Python checks the default path. On UNIX, this default path is

normally /usr/local/lib/python3/.

The module search path is stored in the system module sys as the **sys.path** variable. *The*

*sys.path variable contains the current directory, PYTHONPATH, and the installation dependent default.*

**The PYTHONPATH Variable:**

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax

of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows systemset

PYTHONPATH=c:\python34\lib;

And here is a typical PYTHONPATH from a UNIX systemset

PYTHONPATH=/usr/local/lib/python

**The dir( ) Function**

The dir() built-in function returns a sorted list of strings containing the names defined by

a module.

The list contains the names of all the modules, variables and functions that are defined in

a module.

#!/usr/bin/python3

# Import built-in module math

import math

content = dir(math)

print (content)

When the above code is executed, it produces the following result-

['\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', 'acos', 'asin', 'atan',

'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',

'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',

'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',

'sqrt', 'tan', 'tanh']

Here, the special string variable *\_\_*name*\_\_* is the module's name, and *\_\_*file*\_\_*is the

filename from which the module was loaded.

**The globals() and locals() Functions:**

The **globals() and locals()** functions can be used to return the names in the global and

local namespaces depending on the location from where they are called.

If **locals()** is called from within a function, it will return all the names that can be

accessed locally from that function.

If **globals()** is called from within a function, it will return all the names that can

be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted

using the **keys()** function.

**The reload() Function:**

When a module is imported into a script, the code in the top-level portion of a module is

executed only once.

Therefore, if you want to reexecute the top-level code in a module, you can use

the *reload()* function. The reload() function imports a previously imported module again.

The syntax of the reload() function is this-

reload(module\_name)

**Packages in Python**

A package is a hierarchical file directory structure that defines a single Python application

environment that consists of modules and subpackages and sub-subpackages, and so on.

Consider a file Pots.py available in Phone directory. This file has the following line of

source code-

#!/usr/bin/python3

def Pots():

print ("I'm Pots Phone")

Similarly, we have other two files having different functions with the same name as above.

They are −

*Phone/Isdn.py* file having function Isdn()

*Phone/G3.py* file having function G3()

Now, create one more file \_\_init\_\_.py in the *Phone* directory-

Phone/\_\_init\_\_.py.

To make all of your functions available when you have imported Phone, you need to put

explicit import statements in \_\_init\_\_.py as follows

from Pots import Pots

from Isdn import Isdn

from G3 import G3

After you add these lines to \_\_init\_\_.py, you have all of these classes available when you

import the Phone package.

#!/usr/bin/python3

# Now import your Phone Package.

import Phone

Phone.Pots()

Phone.Isdn()

Phone.G3()

When the above code is executed, it produces the following result-

I'm Pots Phone

I'm 3G Phone

I'm ISDN Phone

In the above example, we have taken example of a single function in each file, but you

can keep multiple functions in your files. You can also define different Python classes in

those files and then you can create your packages out of those classes.

1. **Iterators:**

Iterator allows us to traverse through the elements in collections( List, Dictionary, Array). It uses two method mainly – iter(), next().

Through iter() we create the iterator object and through next() we traverse through the elements.

String, List or Tuple objects can be used to create an Iterator.

list = [1,2,3,4,5]

it = iter(list) # this builds an iterator object

print (next(it)) # prints next available element in iterator

*Iterator object can be traversed using regular for statement*

!usr/bin/python3

for x in it:

print (x, end=" ")

*using next() function:*

while True:

try:

print (next(it))

except StopIteration:

sys.exit() #you have to import sys module for this

1. **Generator:**

A generator is a function that produces or yields a sequence of values using yield method.

When a generator function is called it only generate the generator object without executing it .When the next() method is called then only the Generator function start its first execution. The yield keeps track i.e. remembers the last execution and the second next() call continues from previous value.

import sys

def fibonacci(n): #generator function

a, b, counter = 0, 1, 0

while True:

if (counter > n):

return

yield a

a, b = b, a + b

counter += 1

f = fibonacci(5) #f is generator(iterator) object

while True:

try:

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

except StopIteration:

sys.exit()

**Write an empty function in Python – pass statement:**

In python we cannot write any blank function like C or JAVA. or it will create an error. To define a empty function we have to define the function using Pass statement.

# Correct way of writing empty function

# in Python

def fun():

    pass

# Empty loop in Python

mutex = True

while (mutex == True) :

    pass

# Empty in if/else in Python

mutex = True

if (mutex == True) :

    pass

else :

    print("False")

**Yield vs Return:**

The yield statement suspends function’s execution and sends a value back to caller, but retains enough state to enable function to resume where it is left off. When resumed, the function continues execution immediately after the last yield run. This allows its code to produce a series of values over time, rather them computing them at once and sending them back like a list.

So we can say Yield does not give the output once at a time. It executes step by step and give the outputs one by one. Return compute all the values one at a time and give the output in List fashion way.

**Return** sends a specified value back to its caller whereas **Yield** can produce a sequence of values. We should use yield when we want to iterate over a sequence, but don’t want to store the entire sequence in memory.

Yield are used in Python **generators**. A generator function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function.

# A Simple Python program to demonstrate working

# of yield

# A generator function that yields 1 for first time,

# 2 second time and 3 third time

def simpleGeneratorFun():

    yield 1

    yield 2

    yield 3

# Driver code to check above generator function

for value in simpleGeneratorFun():

    print(value)

# A Python program to generate squares from 1

# to 100 using yield and therefore generator

# An infinite generator function that prints

# next square number. It starts with 1

def nextSquare():

    i = 1;

    # An Infinite loop to generate squares

    while True:

        yield i\*i

        i += 1  # Next execution resumes

                # from this point

# Driver code to test above generator

# function

for num in nextSquare():

    if num > 100:

         break

    print(num)

**Returning Multiple Values in Python:**

We can return multiple values through different options:

1. Through Tuple
2. Through List
3. Through Dictionary
4. Through Object ( Same as C and JAVA)

**Through Tuple:**

def fun():

str = “sourav”

x = 61

return (str,x);

str,x = fun();

print (str)

print (x)

**Through List:**

def fun():

str = “sourav”

x = 61

return [str,x];

list = fun();

print (list)

**Through Dictionary:**

def fun():

    d = dict();

    d['str'] = "GeeksforGeeks"

    d['x']   = 20

    return d

# Driver code to test above method

d = fun()

print(d)

Output:

{'x': 20, 'str': 'GeeksforGeeks'}

**Through Object:**

class Test:

    def \_\_init\_\_(self):

        self.str = "geeksforgeeks"

        self.x = 20

def fun():

return Test();

t= fun();

print (t.str)

print(t.x)

**\*args and \*\*kwargs in python:**

First of all let me tell you that it is not necessary to write \*args or \*\*kwargs. Only the \* (aesteric) is necessary. You could have also written \*var and \*\*vars. Writing \*args and \*\*kwargs is just a convention. So now lets take a look at \*args first.

\*args and \*\*kwargs are mostly used in function definitions. \*args and \*\*kwargs allow you to pass a variable number of arguments to a function. What does variable mean here is that you do not know before hand that how many arguments can be passed to your function by the user so in this case you use these two keywords.

**Usage of \*args:**

\*args is used to send a ***non-keyworded*** variable length argument list to the function.

Here’s an example to help you get a clear idea:

def sum(\*args):

    s = 0

    for i in args:

        s += i

    print("sum is", s)

>>> sum(1, 2, 3)

6

>>> sum(1, 2, 3, 4, 5, 7)

22

>>> sum(1, 2, 3, 4, 5, 7, 8, 9, 10)

49

>>> sum()

0

def test\_var\_args(f\_arg, \*argv):

print "first normal arg:", f\_arg

for arg in argv:

print "another arg through \*argv :", arg

test\_var\_args('yasoob','python','eggs','test')

**Output**:

first normal arg: yasoob

another arg through \*argv : python

another arg through \*argv : eggs

another arg through \*argv : test

**Usage of \*\*kwargs:**

It is same as \*args but this is used for **keyworded** parameters.

def my\_func(\*\*kwargs):

    for i, j in kwargs.items():

        print(i, j)

my\_func(name='tim', sport='football', roll=19)

Output:

sport football

roll 19

name tim

def greet\_me(\*\*kwargs):

if kwargs is not None:

for key, value in kwargs.iteritems():

print "%s == %s" %(key,value)

>>> greet\_me(name="yasoob")

name == yasoob

**Using \*args and \*\*kwargs to call a function:**

def test\_args\_kwargs(arg1, arg2, arg3):

print "arg1:", arg1

print "arg2:", arg2

print "arg3:", arg3

# first with \*args

>>> args = ("two", 3,5)

>>> test\_args\_kwargs(\*args) # here list is broken into three elements

arg1: two

arg2: 3

arg3: 5

This works only when number of argument is same as number of elements in the iterable variable.

# now with \*\*kwargs:

>>> kwargs = {"arg3": 3, "arg2": "two","arg1":5}

>>> test\_args\_kwargs(\*\*kwargs)

arg1: 5

arg2: two

arg3: 3

**Note**: For this to work 2 things are necessary:

1. Names of arguments in function must match with the name of keys in dictionary.
2. Number of arguments should be same as number of keys in the dictionary.

**Order of using \*args \*\*kwargs and formal args**

Order is:

some\_func(fargs,\*args,\*\*kwargs)

**Code Snippet illustrating python built-in modules:**

# importing built-in module math

import math

# using square root(sqrt) function contained

# in math module

print math.sqrt(25)

# using pi function contained in math module

print math.pi

# 2 radians = 114.59 degreees

print math.degrees(2)

# 60 degrees = 1.04 radians

print math.radians(60)

# Sine of 2 radians

print math.sin(2)

# Cosine of 0.5 radians

print math.cos(0.5)

# Tangent of 0.23 radians

print math.tan(0.23)

# 1 \* 2 \* 3 \* 4 = 24

print math.factorial(4)

# importing built in module random

import random

# printing random integer between 0 and 5

print random.randint(0, 5)

# print random floating point number between 0 and 1

print random.random()

# random number between 0 and 100

print random.random() \* 100

List = [1, 4, True, 800, "python", 27, "hello"]

# using choice function in random module for choosing

# a random element from a set such as a list

print random.choice(List)

# importing built in module datetime

import datetime

from datetime import date

import time

# Returns the number of seconds since the

# Unix Epoch, January 1st 1970

print time.time()

# Converts a number of seconds to a date object

print date.fromtimestamp(454554)

**Mathematical Functions in Python**

<https://www.geeksforgeeks.org/mathematical-functions-python-set-1-numeric-functions/>

|  |  |
| --- | --- |
| **Functions** | **Comment** |
| **ceil()** | This function returns the **smallest integral value greater than the number**. If number is already integer, same number is returned. |
| **floor()** | This function returns the **greatest integral value smaller than the number**. If number is already integer, same number is returned. |
| **fabs()** | This function returns the **absolute value** of the number. |
| **factorial()** | This function returns the **factorial** of the number. An error message is displayed if number is not integral. |
| **copysign(a, b)** | This function returns the number with the **value of ‘a’ but with the sign of ‘b’**. The returned value is float type. |
| **gcd()** | This function is used to compute the **greatest common divisor of 2 numbers**mentioned in its arguments. This function works in python 3.5 and above. |

**Logarithmic and Power Functions**

|  |  |
| --- | --- |
| **Functions** | **Comment** |
| **exp(a)** | This function returns the value of**e raised to the power a (e\*\*a)** |
| **log(a, b)** | This function returns the logarithmic **value of a with base b**. If base is not mentioned, the computed value is of natural log. |
| **log2(a)** | This function computes value of **log a with base 2**. This value is **more accurate**than the value of the function discussed above. |
| **log10(a)** | This function computes value of **log a with base 10**. This value is **more accurate**than the value of the function discussed above. |
| **pow(a, b)** | This function is used to compute value of **a raised to the power b (a\*\*b)**. |
| **sqrt()** | This function returns the **square root** of the number |

**Inplace Operators in Python:**

Python in its definition provides methods to perform inplace operations, i.e **doing assignment and computation in a single statement** using “**operator**” module.

x += y is equivalent to x = operator.iadd(x, y)

# Python code to demonstrate the working of

# iadd() and iconcat()

# importing operator to handle operator operations

import operator

# using iadd() to add and assign value

x = operator.iadd(2, 3);

# printing the modified value

print ("The value after adding and assigning : ", end="")

print (x)

# initializing values

y = "geeks"

z = "forgeeks"

# using iconcat() to concat the sequences

y = operator.iconcat(y, z)

# using iconcat() to concat sequences

print ("The string after concatenation is : ", end="")

print (y)

# Python code to demonstrate the working of

# isub() and imul()

# importing operator to handle operator operations

import operator

# using isub() to subtract and assign value

x = operator.isub(2, 3);

# printing the modified value

print ("The value after subtracting and assigning : ", end="")

print (x)

# using imul() to multiply and assign value

x = operator.imul(2, 3);

# printing the modified value

print ("The value after multiplying and assigning : ", end="")

print (x)

# using itruediv() to divide and assign value

x = operator.itruediv(10, 5);

# printing the modified value

print ("The value after dividing and assigning : ", end="")

print (x)

# using imod() to modulus and assign value

x = operator.imod(10, 6);

# printing the modified value

print ("The value after modulus and assigning : ", end="")

print (x)

# using ixor() to exclusive or and assign value

x = operator.ixor(10,5);

# printing the modified value

print ("The value after xoring and assigning : ",end="")

print (x)

# using ipow() to exponentiate and assign value

x = operator.ipow(5,4);

# printing the modified value

print ("The value after exponentiating and assigning : ",end="")

print (x)

x = operator.ior(10,5);

# printing the modified value

print ("The value after bitwise or, and assigning : ",end="")

print (x)

# using iand() to and, and assign value

x = operator.iand(5,4);

# printing the modified value

print ("The value after bitwise and, and assigning : ",end="")

print (x)

# using ilshift() to bitwise left shift and assign value

x = operator.ilshift(8,2);

# printing the modified value

print ("The value after bitwise left shift and assigning : ",end="")

print (x)

# using irshift() to bitwise right shift and assign value

x = operator.irshift(8,2);

# printing the modified value

print ("The value after bitwise right shift and assigning : ",end="")

print (x)

**Calendar Functions**

<https://www.geeksforgeeks.org/calendar-functions-in-python-set-1-calendar-month-isleap/>

<https://www.geeksforgeeks.org/calendar-functions-in-python-set-2monthrange-prcal-weekday/>

[**Complex Numbers**](https://www.geeksforgeeks.org/complex-numbers-in-python-set-1-introduction/)

<https://www.geeksforgeeks.org/complex-numbers-in-python-set-1-introduction/>

<https://www.geeksforgeeks.org/complex-numbers-python-set-2-important-functions-constants/>

[**Trigonometric and Hyperbolic Functions**](https://www.geeksforgeeks.org/complex-numbers-in-python-set-3-trigonometric-and-hyperbolic-functions/)**:**

https://www.geeksforgeeks.org/complex-numbers-in-python-set-3-trigonometric-and-hyperbolic-functions/

**Time Functions in Python**

<https://www.geeksforgeeks.org/time-functions-in-python-set-1-time-ctime-sleep/>

https://www.geeksforgeeks.org/time-functions-python-set-2-date-manipulations/

**Random Module:**

Python defines a set of functions that are used to generate or manipulate random numbers. This particular type of functions are used in a lot of games, lotteries or any application requiring random number generation.

**# Python code to demonstrate the working of**

**# choice() and randrange()**

# importing "random" for random operations

import random

# using choice() to generate a random number from a

# given list of numbers.

print ("A random number from list is : ",end="")

print (random.choice([1, 4, 8, 10, 3]))

# using randrange() to generate in range from 20

# to 50. The last parameter 3 is step size to skip

# three numbers when selecting.

print ("A random number from range is : ",end="")

print (random.randrange(20, 50, 3))

Output:

A random number from list is : 4

A random number from range is : 41

**# Python code to demonstrate the working of**

**# random() and seed()**

# importing "random" for random operations

import random

# using random() to generate a random number

# between 0 and 1

print ("A random number between 0 and 1 is : ", end="")

print (random.random())

# using seed() to seed a random number

random.seed(5)

# printing mapped random number

print ("The mapped random number with 5 is : ", end="")

print (random.random())

# using seed() to seed different random number

random.seed(7)

# printing mapped random number

print ("The mapped random number with 7 is : ", end="")

print (random.random())

# using seed() to seed to 5 again

random.seed(5)

# printing mapped random number

print ("The mapped random number with 5 is : ",end="")

print (random.random())

# using seed() to seed to 7 again

random.seed(7)

# printing mapped random number

print ("The mapped random number with 7 is : ",end="")

print (random.random())

Output:

A random number between 0 and 1 is : 0.510721762520941

The mapped random number with 5 is : 0.6229016948897019

The mapped random number with 7 is : 0.32383276483316237

The mapped random number with 5 is : 0.6229016948897019

The mapped random number with 7 is : 0.32383276483316237

**# Python code to demonstrate the working of**

**# shuffle() and uniform()**

# importing "random" for random operations

import random

# Initializing list

li = [1, 4, 5, 10, 2]

# Printing list before shuffling

print ("The list before shuffling is : ", end="")

for i in range(0, len(li)):

    print (li[i], end=" ")

print("\r")

# using shuffle() to shuffle the list

random.shuffle(li)

# Printing list after shuffling

print ("The list after shuffling is : ", end="")

for i in range(0, len(li)):

    print (li[i], end=" ")

print("\r")

# using uniform() to generate random floating number in range

# prints number between 5 and 10

print ("The random floating point number between 5 and 10 is : ",end="")

print (random.uniform(5,10))

Output:

The list before shuffling is : 1 4 5 10 2

The list after shuffling is : 2 1 4 5 10

The random floating point number between 5 and 10 is : 5.183697823553464

**Reduce() in Python**

Reduce function is used to apply a specific function to all the element in a specific seq. The syntax of reduce is :

reduce(func,seq)

This function is defined in “**functools**” module.

1. At first step, first two elements of sequence are picked and the result is obtained.
2. Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.
3. This process continues till no more elements are left in the container.
4. The final returned result is returned and printed on console.

import functools

list = [1,2,3,4,5]

#for Summation

print(functools.reduce(lambda a,b: a+b,list))

#for getting the maximum element from the list

print(functools.reduce(lambda a,b: a if a>b else b,list))

**Using Operator Functions:**

# python code to demonstrate working of reduce()

# using operator functions

# importing functools for reduce()

import functools

# importing operator for operator functions

import operator

# initializing list

lis = [ 1 , 3, 5, 6, 2, ]

# using reduce to compute sum of list

# using operator functions

print ("The sum of the list elements is : ",end="")

print (functools.reduce(operator.add,lis))

# using reduce to compute product

# using operator functions

print ("The product of list elements is : ",end="")

print (functools.reduce(operator.mul,lis))

# using reduce to concatenate string

print ("The concatenated product is : ",end="")

print (functools.reduce(operator.add,["geeks","for","geeks"]))

**reduce() vs accumulate():**

Both reduce() and accumulate() can be used to calculate the summation of a sequence elements. But there are differences in the implementation aspects in both of these.

1. reduce() is defined in “functools” module, accumulate() in “itertools” module.
2. reduce() stores the intermediate result and only returns the final summation value. Whereas, accumulate() returns a list containing the intermediate results. The last number of the list returned is summation value of the list.
3. reduce(fun,seq) takes function as 1st and sequence as 2nd argument. In contrast accumulate(seq,fun) takes sequence as 1st argument and function as 2nd argument.

# python code to demonstrate summation

# using reduce() and accumulate()

# importing itertools for accumulate()

import itertools

# importing functools for reduce()

import functools

# initializing list

lis = [ 1, 3, 4, 10, 4 ]

# priting summation using accumulate()

print ("The summation of list using accumulate is :",end="")

print (list(itertools.accumulate(lis,lambda x,y : x+y)))

# priting summation using reduce()

print ("The summation of list using reduce is :",end="")

print (functools.reduce(lambda x,y:x+y,lis))

Output:

The summation of list using accumulate is :[1, 4, 8, 18, 22]

The summation of list using reduce is :22

**pprint : Data pretty printer in Python**

You can print a list and dictionary in a beautiful format in the Python repl. Here is the relevant code:

from pprint import pprint

my\_dict = {'name': 'Yasoob', 'age': 'undefined', 'personality': 'awesome'}

pprint(my\_dict)

This is more effective on dict s. Moreover, if you want to pretty print json quickly from a file then you can simply do:

cat file.json | python -m json.tool

**Python eval():**

<https://www.programiz.com/python-programming/methods/built-in/eval>

<https://www.geeksforgeeks.org/eval-in-python/>

<https://stackoverflow.com/questions/9383740/what-does-pythons-eval-do>