




# PYTHON



# PYTHON FUNCTIONS



Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code, which can be called whenever required.

Python allows us to divide a large program into the basic building blocks known as a function. The function contains the set of programming statements enclosed by `{}`. A function can be called multiple times to provide reusability and modularity to the Python program.

The Function helps to programmer to break the program into the smaller part. It organizes the code very effectively and avoids the repetition of the code. As the program grows, function makes the program more organized.



Python provide us various inbuilt functions like **range()** or **print()**. Although, the user can create its functions, which can be called user-defined functions.

There are mainly two types of functions.

**User-define functions** - The user-defined functions are those define by the **user** to perform the specific task.

**Built-in functions** - The built-in functions are those functions that are **pre-defined** in Python.

In this tutorial, we will discuss the user define functions.

# ADVANTAGE OF FUNCTIONS IN PYTHON

- Using functions, we can avoid rewriting the same logic/code again and again in a program.
- We can call Python functions multiple times in a program and anywhere in a program.
- We can track a large Python program easily when it is divided into multiple functions.
- Reusability is the main achievement of Python functions.
- However, Function calling is always overhead in a Python program.

# CREATING A FUNCTION

Python provides the **def** keyword to define the function. The syntax of the define function is given below.

```
def my_function([parameters]):  
    function_block  
  
    return expression
```



The **def** keyword, along with the function name is used to define the function.

The identifier rule must follow the function name.

A function accepts the parameter (argument), and they can be optional.

The function block is started with the colon (:), and block statements must be at the same indentation.

The **return** statement is used to return the value. A function can have only one **return**

# FUNCTION CALLING

In Python, after the function is created, we can call it from another function. A function must be defined before the function call; otherwise, the Python interpreter gives an error. To call the function, use the function name followed by the parentheses.

Consider the following example of a simple example that prints the message "Hello World".





#function definition

**def** hello\_world():

**print**("hello world")

# function calling

hello\_world()

# THE RETURN STATEMENT

The return statement is used at the end of the function and returns the result of the function. It terminates the function execution and transfers the result where the function is called. The return statement cannot be used outside of the function.

It can contain the expression which gets evaluated and value is returned to the caller function. If the return statement has no expression or does not exist itself in the function then it returns the **None** object.



# Defining function

**def** sum():

    a = 10

    b = 20

    c = a+b

**return** c

# calling sum() function in print statement

**print**("The sum is:",sum())



# Defining function

**def** sum():


    a = 10

    b = 20

    c = a+b

# calling sum() function in print statement

**print**(sum())



in the above code, we have defined the same function without the return statement as we can see that the **sum()** function returned the **None** object to the caller function.

# ARGUMENTS IN FUNCTION

The arguments are types of information which can be passed into the function. The arguments are specified in the parentheses. We can pass any number of arguments, but they must be separate them with a comma.

#defining the function

```
def func (name):
```

```
    print("Hi ",name)
```

#calling the function

```
func("Devansh")
```

#Python function to calculate the sum of two variables defining the function

**def** sum (a,b):

**return** a+b;

#taking values from the user

a = int(input("Enter a: "))

b = int(input("Enter b: "))

#printing the sum of a and b

**print**("Sum = ",sum(a,b))

# CALL BY REFERENCE IN PYTHON

In Python, call by reference means passing the actual value as an argument in the function. All the functions are called by reference, i.e., all the changes made to the reference inside the function revert back to the original value referred by the reference.



#defining the function

**def** change\_list(list1):

list1.append(20)

list1.append(30)

**print**("list inside function = ",list1)

#defining the list

list1 = [10,30,40,50]

#calling the function

change\_list(list1)

**print**("list outside function = ",list1)

#defining the function

**def** change\_string (str):

    str = str + " Hows you "

**print**("printing the string inside function :",str)

string1 = "Hi I am there"

#calling the function

change\_string(string1)

**print**("printing the string outside function :",string1)


# TYPES OF ARGUMENTS

There may be several types of arguments which can be passed at the time of function call.

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

# REQUIRED ARGUMENTS

Till now, we have learned about function calling in Python. However, we can provide the arguments at the time of the function call. As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition. If either of the arguments is not provided in the function call, or the position of the arguments is changed, the Python interpreter will show the error.



```
def func(name):  
    message = "Hi "+name  
    return message  
name = input("Enter the name:")  
print(func(name))
```

#the function simple\_interest accepts three arguments and returns the simple interest accordingly

```
def simple_interest(p,t,r):
```

```
    return (p*t*r)/100
```

```
p = float(input("Enter the principle amount? "))
```

```
r = float(input("Enter the rate of interest? "))
```

```
t = float(input("Enter the time in years? "))
```

```
print("Simple Interest: ",simple_interest(p,r,t))
```



#the function calculate returns the sum of two arguments a and b

**def** calculate(a,b):


**return** a+b

calculate(10) # this causes an error as we are missing a required arguments b.

# DEFAULT ARGUMENTS

Python allows us to initialize the arguments at the function definition. If the value of any of the arguments is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.





```
def printme(name,age=22):  
    print("My name is",name,"and age is",age)  
printme(name = "john")
```



```
def printme(name,age=22):
```

```
    print("My name is",name,"and age is",age)
```


```
printme(name = "john") #the variable age is not passed into the function however the  
default value of age is considered in the function
```

```
printme(age = 10,name="David") #the value of age is overwritten here, 10 will be p  
rinted as age
```


# VARIABLE-LENGTH ARGUMENTS (\*ARGS)

In large projects, sometimes we may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to offer the comma-separated values which are internally treated as tuples at the function call. By using the variable-length arguments, we can pass any number of arguments.

However, at the function definition, we define the variable-length argument using the **\*args** (star) as `*<variable - name >`.



```
def printme(*names):  
    print("type of passed argument is ",type(names))  
    print("printing the passed arguments...")  
    for name in names:  
        print(name)  
printme("john","David","smith","nick")
```



In the above code, we passed **\*names** as variable-length argument. We called the function and passed values which are treated as tuple internally. The tuple is an iterable sequence the same as the list. To print the given values, we iterated **\*arg names** using for loop.

# KEYWORD ARGUMENTS(\*\*KWARGS)

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

The name of the arguments is treated as the keywords and matched in the function calling and definition. If the same match is found, the values of the arguments are copied in the function definition.



#function func is called with the name and message as the keyword arguments

**def** func(name,message):

**print**("printing the message with",name,"and ",message)

    #name and message is copied with the values John and hello respectively

func(name = "John",message="hello")



#The function `simple_interest(p, t, r)` is called with the keyword arguments the order of arguments doesn't matter in this case

```
def simple_interest(p,t,r):
```

```
    return (p*t*r)/100
```

```
print("Simple Interest: ",simple_interest(t=10,r=10,p=1900))
```






#The function `simple_interest(p, t, r)` is called with the keyword arguments.

```
def simple_interest(p,t,r):
```


```
    return (p*t*r)/100
```

# doesn't find the exact match of the name of the arguments (keywords)


```
print("Simple Interest: ",simple_interest(time=10,rate=10,principle=1900))
```



```
def func(name1,message,name2):  
    print("printing the message with",name1,",",message,",and",name2)  
#the first argument is not the keyword argument  
func("John",message="hello",name2="David")
```




```
def func(name1,message,name2):  
    print("printing the message with",name1,",",message,",and",name2)  
func("John",message="hello","David")
```



Python provides the facility to pass the multiple keyword arguments which can be represented as **\*\*kwargs**. It is similar as the **\*args** but it stores the argument in the dictionary format.

This type of arguments is useful when we do not know the number of arguments in advance.



```
def food(**kwargs):
```


```
    print(kwargs)
```

```
food(a="Apple")
```

```
food(fruits="Orange", Vagitables="Carrot")
```



# SCOPE OF VARIABLES



The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.

In python, the variables are defined with the two types of scopes.

1. Global variables
2. Local variables

The variable defined outside any function is known to have a global scope, whereas the variable defined inside a function is known to have a local scope.



```
def print_message():
```

```
    message = "hello !! I am going to print a message." # the variable message is local to the function itself
```

```
    print(message)
```

```
print_message()
```

```
print(message) # this will cause an error since a local variable cannot be accessible here.
```





```
def calculate(*args):
```

```
    sum=0
```

```
    for arg in args:
```

```
        sum = sum +arg
```

```
    print("The sum is",sum)
```


```
sum=0
```

```
calculate(10,20,30) #60 will be printed as the sum
```

```
print("Value of sum outside the function:",sum) # 0 will be printed Output:
```



# PYTHON BUILT-IN FUNCTIONS



The Python built-in functions are defined as the functions whose functionality is pre-defined in Python. The python interpreter has several functions that are always present for use. These functions are known as Built-in Functions. There are several built-in functions in Python which are listed below:

# ABS()

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



```
# integer number
```

```
integer = -20
```

```
print('Absolute value of -40 is:', abs(integer))
```

# ALL()

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.



```
# all values true
```

```
k = [1, 3, 4, 6]
```

```
print(all(k))
```

```
# all values false
```

```
k = [0, False]
```

```
print(all(k))
```

```
# one false value
```

```
k = [1, 3, 7, 0]
```

```
print(all(k))
```



# one true value

k = [0, False, 5]

**print**(all(k))

# empty iterable

k = []

**print**(all(k))



# BIN()

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


```
x = 10
```

```
y = bin(x)
```

```
print (y)
```

# BOOL()

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



```
test1 = []
```

```
print(test1,'is',bool(test1))
```

```
test1 = [0]
```

```
print(test1,'is',bool(test1))
```

```
test1 = 0.0
```

```
print(test1,'is',bool(test1))
```


```
test1 = None
```

```
print(test1,'is',bool(test1))
```

# 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.



```
string = "Hello World."  
array = bytes(string, 'utf-8')  
print(array)
```

# CALLABLE()

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




```
x = 8
```

```
print(callable(x))
```

# COMPILE()

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






```
# compile string source to code
code_str = 'x=5\ny=10\nprint("sum =",x+y)'
code = compile(code_str, 'sum.py', 'exec')
print(type(code))
exec(code)
exec(x)
```

# EXEC()

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.



```
x = 8
```

```
exec('print(x==8)')
```

```
exec('print(x+4)')
```

# SUM()

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

```
s = sum([1, 2, 4 ])
```


```
print(s)
```

```
s = sum([1, 2, 4], 10)
```

```
print(s)
```

# ANY()

The python **any()** function returns true if any item in an iterable is true. Otherwise, it returns False.



```
l = [4, 3, 2, 0]
```

```
print(any(l))
```

```
l = [0, False]
```

```
print(any(l))
```

```
l = [0, False, 5]
```

```
print(any(l))
```

```
l = []
```

```
print(any(l))
```

# ASCII()

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.



```
normalText = 'Python is interesting'
```

```
print(ascii(normalText))
```

```
otherText = 'Pythön is interesting'
```

```
print(ascii(otherText))
```

```
print('Pyth\xf6n is interesting')
```



# 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.

```
string = "Python is a programming language."
```

```
# string with encoding 'utf-8'
```

```
arr = bytearray(string, 'utf-8')
```

```
print(arr)
```

# EVAL()

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

```
x = 8
```

```
print(eval('x + 1'))
```

# FLOAT()

The python **float()** function returns a floating-point number from a number or string.

```
print(float(9))
```

```
print(float(8.19))
```

```
print(float("-24.27"))
```

```
print(float("    -17.19\n"))
```

```
print(float("xyz"))
```

# FORMAT()

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

# integer

```
print(format(123, "d"))
```

```
print(format(123.4567898, "f"))
```

```
print(format(12, "b"))
```

# FROZENSET()

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

```
# tuple of letters
```

```
letters = ('m', 'r', 'o', 't', 's')
```

```
fSet = frozenset(letters)
```

```
print('Frozen set is:', fSet)
```

```
print('Empty frozen set is:', frozenset())
```

# GETATTR()

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

```
class Details:
```

```
    age = 22
```

```
    name = "Phill"
```

```
details = Details()
```

```
print('The age is:', getattr(details, "age"))
```

```
print('The age is:', details.age)
```

# GLOBALS()

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.

```
age = 22
```

```
globals()['age'] = 22
```

```
print('The age is:', age)
```

# HASATTR()

The python **any()** function returns true if any item in an iterable is true, otherwise it returns False.

```
l = [4, 3, 2, 0]
```

```
print(any(l))
```

```
l = [0, False]
```

```
print(any(l))
```

```
l = [0, False, 5]
```

```
print(any(l))
```

```
l = []
```

```
print(any(l))
```



# ITER()

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



```
# list of numbers
```

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

```
listlter = iter(list)
```

```
# prints '1'
```

```
print(next(listlter))
```

```
# prints '2'
```

```
print(next(listlter))
```

```
# prints '3'
```

```
print(next(listlter))
```

# LEN()

The python **len()** function is used to return the length (the number of items) of an object.

```
strA = 'Python'
```

```
print(len(strA))
```

# LIST()

The python **list()** creates a list in python.

```
# empty list
```

```
print(list())
```

```
String = 'abcde'
```

```
print(list(String))
```

```
Tuple = (1,2,3,4,5)
```

```
print(list(Tuple))
```

```
List = [1,2,3,4,5]
```

```
print(list(List))
```

# LOCALS()

The python **locals()** method updates and returns the dictionary of the 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.



```
def localsAbsent():  
    return locals()
```

```
def localsPresent():  
    present = True  
    return locals()
```

```
print('localsNotPresent:', localsAbsent())  
print('localsPresent:', localsPresent())
```

# MAP()

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.).



```
def calculateAddition(n):
```

```
    return n+n
```

```
numbers = (1, 2, 3, 4)
```

```
result = map(calculateAddition, numbers)
```

```
print(result)
```

```
# converting map object to set
```


```
numbersAddition = set(result)
```

```
print(numbersAddition)
```



# MEMORYVIEW()

The python **memoryview()** function returns a memoryview object of the given argument.



```
#A random bytearray
randomByteArray = bytearray('ABC', 'utf-8')
mv = memoryview(randomByteArray)
# access the memory view's zeroth index
print(mv[0])
# It create byte from memory view
print(bytes(mv[0:2]))
# It create list from memory view
print(list(mv[0:3]))
```

# 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.

```
python = object()
```

```
print(type(python))
```

```
print(dir(python))
```

# CHR()

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.



```
# Calling function
```

```
result = chr(102) # It returns string representation of a char
```

```
result2 = chr(112)
```

```
# Displaying result
```

```
print(result)
```


```
print(result2)
```

```
# Verify, is it string type?
```

```
print("is it string type:", type(result) is str)
```

# 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.



```
# Python complex() function example
```

```
# Calling function
```

```
a = complex(1) # Passing single parameter
```

```
b = complex(1,2) # Passing both parameters
```

```
# Displaying result
```

```
print(a)
```

```
print(b)
```

# DELATTR()

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.





```
class Student:
```

```
    id = 101
```

```
    name = "Pranshu"
```

```
    email = "pranshu@abc.com"
```

```
# Declaring function
```

```
    def getinfo(self):
```

```
        print(self.id, self.name, self.email)
```

```
s = Student()
```

```
s.getinfo()
```

```
delattr(Student,'course') # Removing attribute which is not available
```

```
s.getinfo() # error: throws an error
```

# DIR()

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.



```
# Calling function
```

```
att = dir()
```

```
# Displaying result
```

```
print(att)
```

# DIVMOD()

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

# Python divmod() function example

# Calling function

```
result = divmod(10,2)
```

# Displaying result

```
print(result)
```

# ENUMERATE()

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.

# Calling function

```
result = enumerate([1,2,3])
```


# Displaying result

```
print(result)
```

```
print(list(result))
```

# FILTER()

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**.



# Python filter() function example

**def** filterdata(x):

**if** x>5:

**return** x

# Calling function

result = filter(filterdata,(1,2,6))

# Displaying result

**print**(list(result))

# HASH()

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:





# Calling function

```
result = hash(21) # integer value
```

```
result2 = hash(22.2) # decimal value
```

# Displaying result

```
print(result)
```

```
print(result2)
```

# HELP()

Python **help()** function is used to get help related to the object passed during the call. It takes an optional parameter and returns help information. If no argument is given, it shows the Python help console. It internally calls python's help function.

# Calling function

```
info = help() # No argument
```

# Displaying result

```
print(info)
```

# MIN()

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.



# Calling function

```
small = min(2225,325,2025) # returns smallest element
```

```
small2 = min(1000.25,2025.35,5625.36,10052.50)
```

# Displaying result

```
print(small)
```

```
print(small2)
```

# SET()

In python, a set is a built-in class, and this function is a constructor of this class. 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.



# Calling function

result = set() # empty set

result2 = set('1 2')

result3 = set('javatpoint')

# Displaying result

**print**(result)

**print**(result2)

**print**(result3)

# HEX()

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.



# Calling function

result = hex(1)

# integer value

result2 = hex(342)

# Displaying result

**print**(result)

**print**(result2)



# ID()

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.



```
# Calling function
```

```
val = id("Javatpoint") # string object
```

```
val2 = id(1200) # integer object
```

```
val3 = id([25,336,95,236,92,3225]) # List object
```

```
# Displaying result
```

```
print(val)
```

```
print(val2)
```

```
print(val3)
```

# SLICE()

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.



# Calling function

result = slice(5) # returns slice object

result2 = slice(0,5,3) # returns slice object


# Displaying result

**print**(result)

**print**(result2)

# SORTED()

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 the case of a dictionary, it sorts only keys, not values.



```
str = "javatpoint" # declaring string
# Calling function
sorted1 = sorted(str) # sorting string
# Displaying result
print(sorted1)
```

# NEXT()

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.

```
number = iter([256, 32, 82]) # Creating iterator
```

```
# Calling function
```

```
item = next(number)
```

```
# Displaying result
```

```
print(item)
```

```
# second item
```

```
item = next(number)
```

```
print(item)
```

```
# third item
```

```
item = next(number)
```

```
print(item)
```



# INPUT()

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.

# Calling function

```
val = input("Enter a value: ")
```

# Displaying result

```
print("You entered:",val)
```

# INT()

Python **int()** function is used to get an integer value. It returns an expression converted into an integer number. If the argument is a floating-point, the conversion truncates the number. If the argument is outside the integer range, then it converts the number into a long type.



```
# Calling function
```

```
val = int(10) # integer value
```

```
val2 = int(10.52) # float value
```

```
val3 = int('10') # string value
```

```
# Displaying result
```

```
print("integer values :",val, val2, val3)
```

# OCT()

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.

```
# Calling function
```

```
val = oct(10)
```

```
# Displaying result
```

```
print("Octal value of 10:",val)
```

# ORD()

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

```
print(ord('8')) # Code point of an integer
```

```
print(ord('R')) # Code point of an alphabet
```

```
print(ord('&')) # Code point of a character
```

# POW()

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



```
# positive x, positive y (x**y)
```

```
print(pow(4, 2))
```

```
# negative x, positive y
```

```
print(pow(-4, 2))
```

```
# positive x, negative y (x**-y)
```

```
print(pow(4, -2))
```

```
# negative x, negative y
```

```
print(pow(-4, -2))
```

# PRINT()

The python **print()** function prints the given object to the screen or other standard output devices.

```
print("Python is programming language.")
```

```
x = 7
```

```
# Two objects passed
```

```
print("x =", x)
```

```
y = x
```

```
# Three objects passed
```

```
print('x =', x, '= y')
```



# RANGE()

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.



# empty range

**print**(list(range(0)))

# using the range(stop)

**print**(list(range(4)))

# using the range(start, stop)

**print**(list(range(1,7 )))

# REVERSED()

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

# for string

String = 'Java'

**print**(list(reversed(String)))

# for tuple

Tuple = ('J', 'a', 'v', 'a')

**print**(list(reversed(Tuple)))



# for range

Range = range(8, 12)

**print**(list(reversed(Range)))

# for list

List = [1, 2, 7, 5]

**print**(list(reversed(List)))

# ROUND()

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

# for integers

```
print(round(10))
```

# for floating point

```
print(round(10.8))
```

# even choice

```
print(round(6.6))
```


# STR()

The python **str()** converts a specified value into a string.

```
str('4')
```

# TYPE()

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



```
List = [4, 5]
```

```
print(type(List))
```

```
Dict = {4: 'four', 5: 'five'}
```

```
print(type(Dict))
```

```
class Python:
```

```
    a = 0
```

```
InstanceOfPython = Python()
```

```
print(type(InstanceOfPython))
```



# VARS()

The python **vars()** function returns the `__dict__` attribute of the given object.

**class** Python:

```
def __init__(self, x = 7, y = 9):
```

```
    self.x = x
```


```
    self.y = y
```

```
InstanceOfPython = Python()
```

```
print(vars(InstanceOfPython))
```

# ZIP()

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.



```
numList = [4,5, 6]
```

```
strList = ['four', 'five', 'six']
```

```
# No iterables are passed
```

```
result = zip()
```

```
# Converting iterator to list
```

```
resultList = list(result)
```

```
print(resultList)
```



# Two iterables are passed

```
result = zip(numList, strList)
```


# Converting iterator to set

```
resultSet = set(result)
```

```
print(resultSet)
```



# PYTHON LAMBDA FUNCTIONS



Python Lambda function is known as the anonymous function that is defined without a name. Python allows us to not declare the function in the standard manner, i.e., by using the **def** keyword. Rather, the anonymous functions are declared by using the **lambda** keyword. However, Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.

The anonymous function contains a small piece of code. It simulates inline functions of C and C++, but it is not exactly an inline function.



The syntax to define an anonymous function is given below.

**lambda** arguments: expression

It can accept any number of arguments and has only one expression. It is useful when the function objects are required.



# a is an argument and a+10 is an expression which got evaluated and returned.

x = **lambda** a:a+10


# Here we are printing the function object

**print**(x)

**print**("sum = ",x(20))



In the above example, we have defined the **lambda a: a+10** anonymous function where **a** is an argument and **a+10** is an expression. The given expression gets evaluated and returned the result. The above lambda function is same as the normal function.



```
def x(a):  
    return a+10  
print(sum = x(10))
```

# USE LAMBDA FUNCTION WITH FILTER()

The Python built-in **filter()** **function** accepts a function and a list as an argument. It provides an effective way to filter out all elements of the sequence. It returns the new sequence in which the function evaluates to **True**.



```
#program to filter out the tuple which contains odd numbers
```

```
lst = (10,22,37,41,100,123,29)
```

```
oddlist = tuple(filter(lambda x:(x%3 == 0),lst)) # the tuple contains all the items of t  
he tuple for which the lambda function evaluates to true
```

```
print(oddlist)
```

# USING LAMBDA FUNCTION WITH MAP()

The **map()** function in Python accepts a function and a list. It gives a new list which contains all modified items returned by the function for each item.



```
#program to filter out the list which contains odd numbers
```

```
lst = (10,20,30,40,50,60)
```

```
square_list = list(map(lambda x:x**2,lst)) # the tuple contains all the items of the list  
for which the lambda function evaluates to true
```

```
print(square_tuple)
```



**CONTINUE IN NEXT UNIT . . . .**