1. Functions in Python

Function is a block of code written to carry out a specified task. Functions provide better modularity and a high degree of code reusing.

- You can Pass Data(input) known as parameter to a function
- A Function may or may not return any value(Output)

There are three types of functions in Python:

I. **Built-in functions** The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

II. **User-Defined Functions (UDFs):** The Functions defined by User is known as User Defined Functions. These are defined with the keyword **def**

III. **Anonymous functions,** which are also called lambda functions because they are **not declared** with the standard def keyword.

Functions vs Methods : A method refers to a function which is part of a class. You access it with an instance or object of the class. A function doesn't have this restriction: it just refers to a standalone function. This means that all methods are functions, but not all functions are methods.

1.1 Built-in functions

Built-in Functions				
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	
delattr()	help()	next()	setattr()	
dict()	hex()	object()	slice()	
dir()	id()	oct()	sorted()	

1.2 User-Defined Functions (UDFs):

Following are the rules to define a **U**ser **D**efine **F**unction in Python.

- Function begin with the keyword **def** followed by the function name and parentheses ( ) .
- Any list of parameter(s) or argument(s) should be placed within these parentheses.
- The first statement within a function is the **documentation string** of the function or ***docstring*** is an optional statement
- The **function block** within every function starts with a colon (:) and is indented.
- 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.

Syntax

```
def functionName( list of parameters ):
   "_docstring"
   function_block
   return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example for Creating a Function without parameter

In Python a function is defined using the `def` keyword:

```
>>> def MyMsg1():
  print("Learning to create function")
```

Example for Creating a Function parameter

The following function takes a string as parameter and prints it on screen.

docString

```
def MyMsg2( name ):
   "This prints a passed string into this function"
    print (name ,' is learning to define Python Function')
   return
```

Calling a Function To call a function, use the function name followed by parenthesis:

```
>>> MyMsg1()
```

Calling function MyMsg1

```
Learning to create function
```

Output

```
>>> MyMsg2('Divyaditya')
>>> MyMsg2('Manasvi')
```

Calling Function MyMsg2() twice with different parameter

```
Divyaditya is learning to define Python Function
Manasvi is learning to define Python Function
```

Output

2. Parameter (argument) Passing

We can define UDFs in one of the following ways in Python

1. Function with no argument and no Return value [like MyMsg1(),Add()]
2. Function with no argument and with Return value
3. Python Function with argument and No Return value [like MyMsg2()]
4. Function with argument and Return value

```
# Python Function with No Arguments, and No Return Value  FUNCTION 1
def Add1():
    a = 20
    b = 30
    Sum = a + b
    print("After Calling the Function:", Sum)
Add1()
```

Here we are not passing any parameter to function instead values are assigned within the function and result is also printed within the function . It is not returning any value

```
# Python Function with Arguments, and No Return Value FUNCTION 2
def Add2(a,b):
    Sum = a + b
        print("Result:", Sum)
Add2(20,30)
```

Here we are passing 2 parameters a,b to the function and function is calculating sum of these parameter and result is printed within the function . It is not returning any value

```
# Python Function with Arguments, and Return Value FUNCTION 3
def Add3(a,b):
    Sum = a + b
    Return Sum

Z=Add3(10,12)
print("Result " Z)
```

Here we are passing 2 parameters a,b to the function and function is calculating sum of these parameter and result is returned to the calling statement which is stored in the variable Z

```
# Python Function with No Arguments, and  Return Value FUNCTION 4
def Add4():
    a=11
    b=20
    Sum = a + b
    Return Sum

Z=Add3(10,12)
print("Result " Z)
```

Here we are not passing any parameter to function instead values are assigned within the function but result is returned.

3. Scope : Scope of a Variable or Function may be Global or Local

3.1 Global and Local Variables in Python

Global variables are the one that are defined and declared outside a function and we can use them anywhere.

Local variables are the one that are defined and declared inside a function/block and we can use them only within that function or block

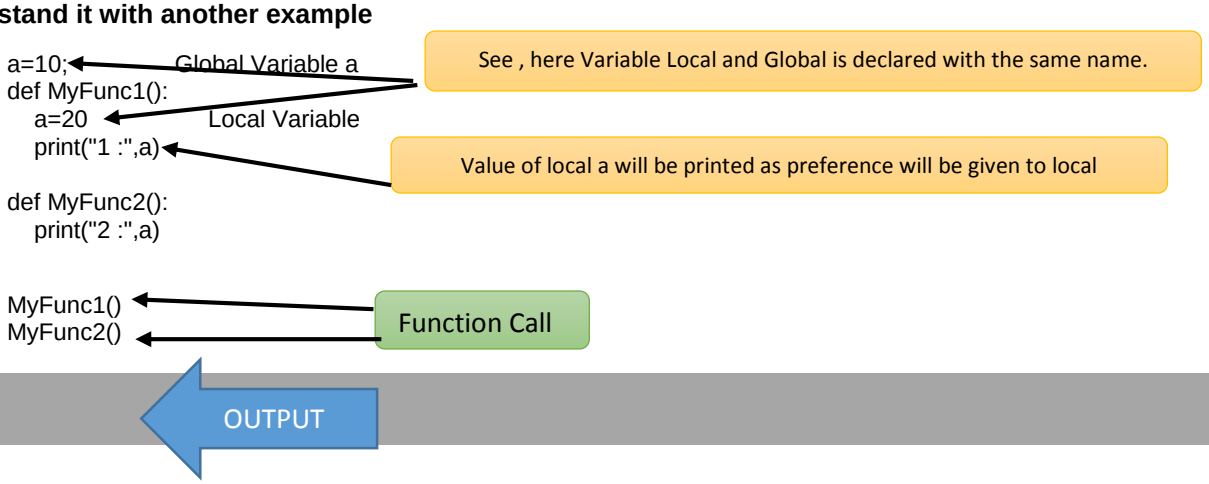
```
A=50

def MyFunc():
   print("Function Called :",a)

MyFunc()
```

Global Varialble

```
Function Called : 50
```

OUTPUT

Lets understand it with another example




```
1 : 20
2 : 10
```

OUTPUT

3.2 Local /Global Functions

Global Functions are the one that are defined and declared outside a function/block and we can use them anywhere.

Local Function are the one that are defined and declared inside a function/block and we can use them only within that function/block

```
a=10;
def MyFunc1():                  # Function is globally defined
    a=20
    print("1 :",a)

def MyFunc2():
    print("2 :",a)
    def SubFun1(st):                                    # Function is Locally defined
        print("Local Function with ",st)
    SubFun1("Local Call")               Function is called Locally
MyFunc1()
MyFunc2()
SubFun1("Global Call")      Function is called Globally will give error as function scope is within the function MyFun2()
```

```
1 : 20
2 : 10
Local Function with Local Call
Traceback (most recent call last):
  File "C:/Users/kv3/AppData/Local/Programs/Python/Python36-32/funct.py", line 14, in <module>
    SubFun1("Global Call")
NameError: name 'SubFun1' is not defined
```

4. Mutable vs Immutable Objects in Python

Every variable in python holds an instance of an object. There are two types of objects in python i.e. **Mutable** and **Immutable objects**. Whenever an object is instantiated, it is assigned a unique object id. The type of the object is defined at the runtime and it can't be changed afterwards. However, it's state can be changed if it is a mutable object.
To summaries the difference, mutable objects can change their state or contents and immutable objects can't change their state or content.

- **Immutable Objects :** These are of in-built types like **int, float, bool, string, unicode, tuple**. In simple words, an immutable object can't be changed after it is created.

```
# Python code to test that
# tuples are immutable

tuple1 = (10, 21, 32, 34)
tuple1[0] = 41
print(tuple1)
```

Error :

```
Traceback (most recent call last):
  File "e0eaddff843a8695575daec34506f126.py", line 3, in
    tuple1[0]=41
TypeError: 'tuple' object does not support item assignment
```

```
# Python code to test that
# strings are immutable

message = "Welcome to Learn Python"
message[0] = 'p'
print(message)
```

Error :

```
Traceback (most recent call last):
  File "/home/ff856d3c5411909530c4d328eeca165b.py", line 3, in
    message[0] = 'w'
TypeError: 'str' object does not support item assignment
```

- **Mutable Objects :** These are of type **list**, **dict**, **set** . Custom classes are generally mutable.

```
# Python code to test that
# lists are mutable
color = ["red", "blue", "green"]
print(color)
color[0] = "pink"
color[-1] = "orange"
print(color)
```

Output:

- `['red', 'blue', 'green']`
- `['pink', 'blue', 'orange']`

Conclusion

1. Mutable and immutable objects are handled differently in python. Immutable objects are fast to access and are expensive to **change**, because it involves creation of a copy. Whereas mutable objects are easy to change.
2. Use of mutable objects is recommended when there is a need to change the size or content of the object.
3. **Exception :** However, there is an exception in immutability as well. We know that tuple in python is immutable. But the tuple consist of a sequence of names with unchangeable bindings to objects.
   Consider a tuple

```
tup = ([3, 4, 5], 'myname')
```

The tuple consist of a string and a list. Strings are immutable so we can't change it's value. But the contents of the list can change. **The tuple itself isn't mutable but contain items that are mutable.**

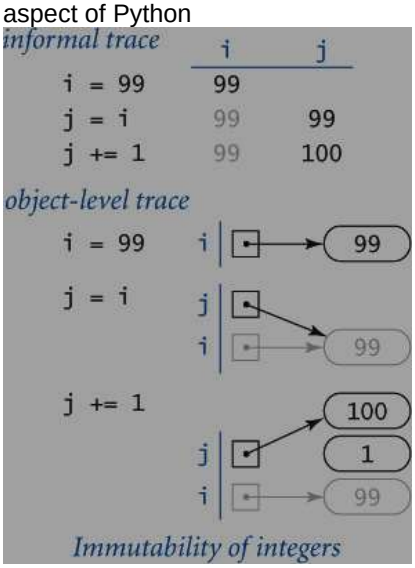
5. Passing arguments and returning values

Reference : Introduction to Programming in Python: An Interdisciplinary Approach By Robert Sedgewick, Robert Dondero, Kevin Wayne

Next, we examine the specifics of Python's mechanisms for passing arguments to and returning values from functions. These mechanisms are conceptually very simple, but it is worthwhile to take the time to understand them fully, as the effects are actually profound. Understanding argument-passing and return-value mechanisms is key to learning *any* new programming language. In the case of Python, the concepts of *immutability* and *aliasing* play a central role.

Call by object reference You can use parameter variables anywhere in the body of the function in the same way as you use local variables. The only difference between a parameter variable and a local variable is that Python initializes the parameter variable with the corresponding argument provided by the calling code. We refer to this approach as call by object reference. (It is more commonly known as call by value, where the value is always an object reference—not the object's value.) One consequence of this approach is that if a parameter variable refers to a mutable object and you change that object's value within a function, then this also changes the object's value in the calling code (because it is the same object). Next, we explore the ramifications of this approach.

Immutability and aliasing Arrays are mutable data types, because we can change array elements. By contrast, a data type is immutable if it is not possible to change the value of an object of that type. The other data types that we have been using (int, float, str, and bool) are all immutable. In an immutable data type, operations that might seem to change a value actually result in the creation of a new object, as illustrated in the simple example at right. First, the statement i = 99 creates an integer99, and assigns to i a reference to that integer. Then j = i assigns i (an object reference) to j, so both i and j reference the same object—the integer 99. Two variables that reference the same objects are said to be aliases. Next, j += 1 results in j referencing an object with value 100, but it does not do so by changing the value of the existing integer from 99 to 100! Indeed, since int objects are immutable, no statement can change the value of that existing integer. Instead, that statement creates a new integer 1, adds it to the integer 99 to create another new integer 100, and assigns to j a reference to that integer. But i still references the original 99. Note that the new integer 1 has no reference to it in the end—that is the system's concern, not ours. The immutability of integers, floats, strings, and booleans is a fundamental aspect of Python

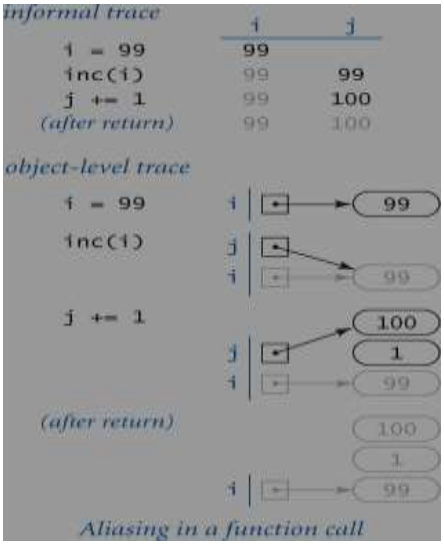


Integers, floats, Booleans, and strings as arguments

The key point to remember about passing arguments to functions in Python is that whenever you pass arguments to a function, the arguments and the function's parameter variables become aliases. In practice, this is the predominant use of aliasing in Python, and it is important to understand its effects. For purposes of illustration, suppose that we need a function that increments an integer (our discussion applies to any more complicated function as well). A programmer new to Python might try this definition:

```
def inc(j):
    j += 1
```

and then expect to increment an integer i with the call inc(i). Code like this would work in some programming languages, but it has no effect in Python, as shown in the figure at right. First, the statement i = 99 assigns to global variable i a reference to the integer 99. Then, the statement inc(i) passes i, an object reference, to the inc() function. That object reference is assigned to the parameter variable j. At this point i and j are aliases. As before, the inc() function's j += 1statement does not change the integer 99, but rather creates a new integer 100 and assigns a reference to that integer to j. But when the inc() function returns to its caller, its parameter variable jgoes out of scope, and the variable i still references the integer 99.



This example illustrates that, in Python, *a function cannot produce the side effect of changing the value of an integer object* (nothing can do so). To increment variable `i`, we could use the definition

```
def inc(j):
    j += 1
    return j
```

and call the function with the assignment statement i = inc(i).
The same holds true for any immutable type. A function cannot change the value of an integer, a float, a boolean, or a string.

Arrays as arguments

When a function takes an array as an argument, it implements a function that operates on an arbitrary number of objects. For example, the following function computes the mean (average) of an array of floats or integers:

```
def mean(a):
    total = 0.0
    for v in a:
        total += v
    return total / len(a)
```

We have been using arrays as arguments from the beginning of the book. For example, by convention, Python collects the strings that you type after the program name in the python command into an arraysys.argv[] and implicitly calls your global code with that array of strings as the argument.

Side effects with arrays

Since arrays are mutable, it is often the case that the purpose of a function that takes an array as argument is to produce a side effect (such as changing the order of array elements). A prototypical example of such a function is one that exchanges the elements at two given indices in a given array. We can adapt the code that we examined at the beginning of SECTION 1.4:

```
def exchange(a, i, j):
   temp = a[i]
   a[i] = a[j]
   a[j] = temp
```

This implementation stems naturally from the Python array representation. The first parameter variable in exchange() is a reference to the array, not to all of the array's elements: when you pass an array as an argument to a function, you are giving it the opportunity to operate on that array (not a copy of it). A formal trace of a call on this function is shown on the facing page. This diagram is worthy of careful study to check your understanding of Python's function-call mechanism.
A second prototypical example of a function that takes an array argument and produces side effects is one that randomly shuffles the elements in the array,

```
def shuffle(a):
    n = len(a)
    for i in range(n):
        r = random.randrange(i, n)
          exchange(a, i, r)
```

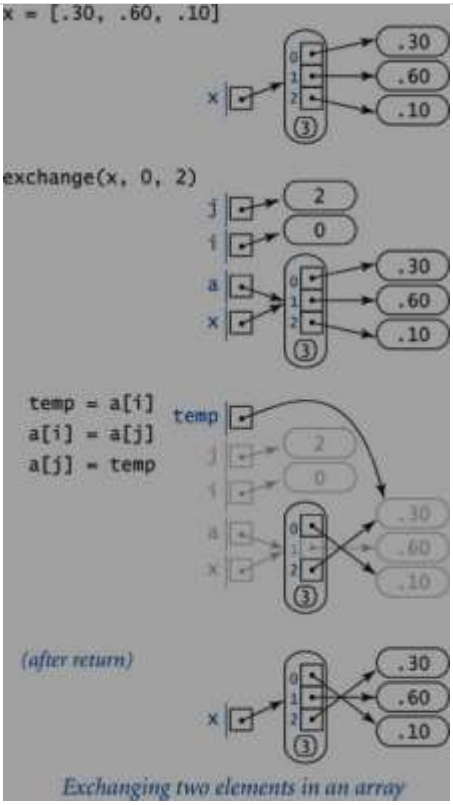
#Python's standard function random.shuffle() does the same task.

Arrays as return values

A function that sorts, shuffles, or otherwise modifies an array taken as argument does not have to return a reference to that array, because it is changing the contents of a client array, not a copy. But there are many situations where it is useful for a function to provide an array as a return value. Chief among these are functions that create arrays for the purpose of returning multiple objects of the same type to a client.

As an example, consider the following function, which returns an array of random floats:

```
def randomarray(n):
    a = stdarray.create1D(n)
    for i in range(n):
        a[i] = random.random()
    return a
```



Exchanging two elements in an array

THE TABLE BELOW CONCLUDES OUR DISCUSSION of arrays as function arguments by highlighting some typical array-procession functions.

mean of an array	`def mean(a):`
	`  total = 0.0`
	`  for v in a:`
	`  total += v`

dot product of two vectors of the same length	`def dot(a, b):`
	`  total = 0`
	`  for i in range(len(a)):`
	`  total += a[i] * b[i]`

exchange two elements in an array	`def exchange(a, i, j):`
	`  temp = a[i]`
	`  a[i] = a[j]`
	`  a[j] = temp`

write a one-dimensional array (and its length)	`def write1D(a):`
	`  stdio.writeln(len(a))`
	`  for v in a:`
	`  stdio.writeln(v)`

read a two-dimensional array of floats (with dimensions)	`def readFloat2D():`
	`  m = stdio.readInt()`
	`  n = stdio.readInt()`
	`  a = stdarray.create2D(m, n, 0.0)`

6. FUNCTION USING LIBRARIEs:

6.1 Functions in Python Math Module

Here is the list of all the functions and attributes defined in `math` module with a brief explanation of what they do.

List of Functions in Python Math Module	
Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
copysign(x, y)	Returns x with the sign of y
fabs(x)	Returns the absolute value of x
factorial(x)	Returns the factorial of x
floor(x)	Returns the largest integer less than or equal to x
fmod(x, y)	Returns the remainder when x is divided by y
frexp(x)	Returns the mantissa and exponent of x as the pair (m, e)
fsum(iterable)	Returns an accurate floating point sum of values in the iterable
isfinite(x)	Returns True if x is neither an infinity nor a NaN (Not a Number)
isinf(x)	Returns True if x is a positive or negative infinity
isnan(x)	Returns True if x is a NaN
ldexp(x, i)	Returns x * (2**i)
modf(x)	Returns the fractional and integer parts of x
trunc(x)	Returns the truncated integer value of x
exp(x)	Returns e**x
expm1(x)	Returns e**x - 1
log(x[, base])	Returns the logarithm of x to the base (defaults to e)
log1p(x)	Returns the natural logarithm of 1+x
log2(x)	Returns the base-2 logarithm of x
log10(x)	Returns the base-10 logarithm of x
pow(x, y)	Returns x raised to the power y
sqrt(x)	Returns the square root of x
acos(x)	Returns the arc cosine of x
asin(x)	Returns the arc sine of x
atan(x)	Returns the arc tangent of x
atan2(y, x)	Returns atan(y / x)
cos(x)	Returns the cosine of x
hypot(x, y)	Returns the Euclidean norm, sqrt(x*x + y*y)
sin(x)	Returns the sine of x
tan(x)	Returns the tangent of x
degrees(x)	Converts angle x from radians to degrees
radians(x)	Converts angle x from degrees to radians
acosh(x)	Returns the inverse hyperbolic cosine of x
asinh(x)	Returns the inverse hyperbolic sine of x
atanh(x)	Returns the inverse hyperbolic tangent of x
cosh(x)	Returns the hyperbolic cosine of x
sinh(x)	Returns the hyperbolic cosine of x
tanh(x)	Returns the hyperbolic tangent of x
erf(x)	Returns the error function at x
erfc(x)	Returns the complementary error function at x
gamma(x)	Returns the Gamma function at x
lgamma(x)	Returns the natural logarithm of the absolute value of the Gamma function at x
pi	Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...)
e	mathematical constant e (2.71828...)

6.1 Python has a set of built-in methods that you can use on strings

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Note: **All string methods returns new values. They do not change the original string.**