FUNCTIONS

Functions are reusable pieces of programs. They allow you to give a name to a block of statements, allowing you to run that block using the specified name anywhere in your program and any number of times. This is known as calling the function. We have already used many built-in functions such as len and range .

Functions are defined using the def keyword. After this keyword comes an identifier name for the function, followed by a pair of parentheses which may enclose some names of variables, and by the final colon that ends the line. Next follows the block of statements that are part of this function.

A function is a block of code which only runs when it is called.
You can pass data, known as parameters, into a function.
A function can return data as a result.

Creating a Function
In Python a function is defined using the def keyword:

Example
```
def my_function():
  print("Hello from a function")
```

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

Example
```
def my_function():
  print("Hello from a function")

my_function()
```

Parameters
Information can be passed to functions as parameter.

Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a function with one parameter (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
def my_function(fname):
  print("Hello "+fname)

my_function("Sachin")
my_function("Sourav")
my_function("Rahul")
```

Function Parameters

A function can take parameters, which are values you supply to the function so that the function can do something utilising those values. These parameters are just like variables except that the values of these variables are defined when we call the function and are already assigned values when the function runs.

Parameters are specified within the pair of parentheses in the function definition, separated by commas. When we call the function, we supply the values in the same way. Note the terminology used - the names given in the function definition are called parameters whereas the values you supply in the function call are called arguments.

```
# Define a function to find max of two given numbers
def print_max(a, b):
    if a > b:
        print a, 'is maximum'
    elif a == b:
        print a, 'is equal to', b
    else:
        print b, 'is maximum'

# End of Function
```

```
# Calling functions
# directly pass literal values
print_max(3, 4)
# pass variables as arguments
x = 5
y = 7
print_max(x, y)
```

Output:

```
4 is maximum
7 is maximum
```

We define a function called print_max that uses two parameters called a and b . We find out the greater number using a simple if..else statement and then print the bigger number. The first time

we call the function print_max , we directly supply the numbers as arguments. In the second case, we call the function with variables as arguments. print_max(x, y) causes the value of argument x to be assigned to parameter a and the value of argument y to be assigned to parameter b . The print_max function works the same way in both cases.

Local Variables

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function - i.e. variable names are local to the function. This is called the scope of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

```
x = 50

# Function definition
def func(x):
    print 'x is', x
    x = 2
    print 'Changed local x to', x

# Calling function
func(x)
print 'x is still', x
```

```
x is 50
Changed local x to 2
x is still 50
```

The first time that we print the value of the name x with the first line in the function's body, Python uses the value of the parameter declared in the main block, above the function definition. Next, we assign the value 2 to x . The name x is local to our function. So, when we change the value of x in the function, the x defined in the main block remains unaffected.

The global statement

If you want to assign a value to a name defined at the top level of the program (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not local, but it is global. We do this using the global statement. It is impossible to assign a value to a variable defined outside a function without the global statement.

```
x = 50

# Function definition
def func():
    global x
    print 'x is', x
    x = 2
    print 'Changed local x to', x

# Calling function
func()
print 'x is changed to ', x
```

```
x is 50
Changed local x to 2
x is changed to  2
```

You can not have same variable act as local as well as global. For example, below program will throw up error as x is defined locally (passed as parameter) and also globally.

```
# Function definition
def func(x):
    global x
    print 'x is', x
    x = 2
    print 'Changed local x to', x

# Calling function
func(x)
print 'x is still', x
```

```
    def func(x):
SyntaxError: name 'x' is local and global
```

Function Arguments

You can call a function by using the following types of formal arguments −

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Default Parameter Value

For some functions, you may want to make some parameters optional and use default values in case the user does not want to provide values for them. This is done with the help of default argument values. You can specify default argument values for parameters by appending to the parameter name in the function definition the assignment operator (=) followed by the default value.

The following example shows how to use a default parameter value.

If we call the function without parameter, it uses the default value:

Example

```
def my_function(country = "India"):
  print("I am from " + country)

my_function("Russia")
my_function()
my_function("Mexico")
my_function()
```

Keyword Arguments

If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called keyword arguments - we

use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function. There are two advantages - one, using the function is easier since we do not need to worry about the order of the arguments. Two, we can give values to only those parameters to which we want to, provided that the other parameters have default argument values.

```
###########  Below example for Default and Keyword Arguments  ####

# define function
def func(a, b=5, c=10):
    print 'a is', a, 'and b is', b, 'and c is', c

# Call functions
func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

```
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

VarArgs parameters

Sometimes you might want to define a function that can take any number of parameters, i.e. variable number of arguments, this can be achieved by using the stars.

When we declare a starred parameter such as *param, then all the positional arguments from that point till the end are collected as a tuple called 'param'.

Similarly, when we declare a double-starred parameter such as **param, then all the keyword arguments from that point till the end are collected as a dictionary called 'param'.

We will explore tuples and dictionaries in a later chapter.

```
#########   Example of Variable Arguments & Return Statement

def total(initial=5, *numbers, **keywords):
    count = initial
    for number in numbers:
        count += number
    for key in keywords:
        count += keywords[key]
    return count

# Call the function
print total(10, 1, 2, 3, vegetables=50, fruits=100)
```

Output:

```
166
```

Return Values
To let a function return a value, use the return statement:

Example
```
def my_function(x):
  return x**2

print(my_function(2))
print(my_function(3))
print(my_function(4))
```

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist.append([1,2,3,4]);
   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
```

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −

```
Values inside the function:  [10, 20, 30, [1, 2, 3, 4]]
Values outside the function:  [10, 20, 30, [1, 2, 3, 4]]
```

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4]; # This would assig 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
The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce the following result −

Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]

DocStrings
Python has a nifty feature called documentation strings, usually referred to by its shorter name docstrings.

```
def print_max(x, y):
    '''Prints the maximum of two numbers.
    The two values must be integers.'''
# convert to integers, if possible
    x = int(x)
    y = int(y)
    if x > y:
        print x, 'is maximum'
    else:
        print y, 'is maximum'


print_max(3, 5)
print print_max.__doc__
```

```
5 is maximum
Prints the maximum of two numbers.
    The two values must be integers.
```

DocStrings are an important tool that you should make use of since it helps to document the program better and makes it easier to understand. We can even get the docstring back from, say a function, when the program is actually running. The convention followed for a docstring is a multi-line string where the first line starts with a capital letter and ends with a dot. Then the second line is blank followed by any detailed explanation starting from the third line. You are strongly advised to follow this convention for all your docstrings for all your non-trivial functions. We can access the docstring of the print_max function using the doc (notice the double underscores) attribute (name

belonging to) of the function. Just remember that Python treats everything as an object and this includes functions.

RECURSIVE FUNCTIONS

Recursion is a method of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition satisfies the condition of recursion, we call this function a recursive function.

Termination condition:
A recursive function has to fulfil an important condition to be used in a program: it has to terminate. A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case. A base case is a case, where the problem can be solved without further recursion. A recursion can end up in an infinite loop, if the base case is not met in the calls.

Example:
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1

Replacing the calculated values gives us the following expression

4! = 4 * 3 * 2 * 1

Let's try a program to implement Factorial of a number using recursion. To see the intermediate steps, we can track how the function works by adding two print() functions to the previous function definition:

```
def factorial(n):
    print("factorial has been called with n = " + str(n))
    if n == 1:
        return 1
    else:
        res = n * factorial(n-1)
        print("track ", n, " * factorial(" ,n-1, "): ",res)
        return res

# Now test the function
print(factorial(5))
```

Let's have a look at an iterative version of the factorial function.

```
def iterative_factorial(n):
    result = 1
    for i in range(2,n+1):
```

```
        result *= i
    return result
```

Lets try one more example, to generate Fibonacci numbers series.

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Iterative version of the program is given below:

```
def fibi(n):
    old, new = 0, 1
    if n == 0:
        return 0
    for i in range(n-1):
        old, new = new, old + new
    return new
```

Some more programs to practice for you. Answers are available in the chapter 15.

1. Write a recursive Python function that returns the sum of the first n integers.
2. Think of a recusive version of the function f(n) = 3 * n, i.e. the multiples of 3
3. Write a function which implements the Pascal's triangle:

```
   1
  1 1
 1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

4. The sum of the squares of two consecutive Fibonacci numbers is also a Fibonacci number, e.g. 2 and 3 are elements of the Fibonacci sequence and 2*2 + 3*3 = 13 corresponds to Fib(7).

Python Lambda

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.

Syntax
lambda arguments : expression
The expression is executed and the result is returned:

Example
A lambda function that adds 10 to the number passed in as an argument, and print the result:

```
x = lambda a : a + 10
print(x(5))
```

Lambda functions can take any number of arguments:

Example
A lambda function that multiplies argument a with argument b and print the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Example
A lambda function that sums argument a, b, and c and print the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Why Use Lambda Functions?
The power of lambda is better shown when you use them as an anonymous function inside another function.

Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

Or, use the same function definition to make a function that always triples the number you send in:

Example
```
def myfunc(n):
  return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

Or, use the same function definition to make both functions, in the same program:

Example
```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

map (), filter (), reduce () functions
These concepts have been covered under LIST. Please refer List Tutorial

DECORATORS

A decorator in Python is any callable Python object that is used to modify a function or a class. The usage of decorates is very easy and powerful, but writing decorators can be complicated, especially if you are not experienced with decorators and some functional programming concepts. Even though it is the same underlying concept, we have two different kinds of decorators in Python:

- Function decorators
- Class decorators

```
def f():
    def g():
        print("Hi, it's me 'g'")
        print("Thanks for calling me")

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

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

f()
```

Output:

This is the function 'f'

I am calling 'g' now:

Hi, it's me 'g'

Thanks for calling me

Another example:

```
def temperature(t):
    def celsius2fahrenheit(x):
        return 9 * x / 5 + 32

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

print(temperature(20))
```

Functions as Parameters

Above examples didn't create any interest in decorators? Then how about this one:

```
def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")

def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()

f(g)
```

Output:

```
Hi, it's me 'f'

I will call 'func' now

Hi, it's me 'g'

Thanks for calling me
```

Due to the fact that every parameter of a function is a reference to an object and functions are objects as well, we can pass functions - or better "references to functions" - as parameters to a function.
We need to know what the 'real' name of func is. For this purpose, we can use the attribute __name__, as it contains this name:

```
def g():
    print("Hi, it's me 'g'")
    print("Thanks for calling me")
def f(func):
    print("Hi, it's me 'f'")
    print("I will call 'func' now")
    func()
    print("func's real name is " + func.__name__)

f(g)
```

Output:

```
Hi, it's me 'f'

I will call 'func' now

Hi, it's me 'g'

Thanks for calling me

func's real name is g
```

Another example:

```
import math

def foo(func):
    print("The function " + func.__name__ + " was passed to foo")
    res = 0
    for x in [1, 2, 2.5]:
        res += func(x)
```

```
    return res

print(foo(math.sin))
print(foo(math.cos))
```

Functions returning Functions

The output of a function is also a reference to an object. Therefore functions can return references to function objects.

```
def f(x):
    def g(y):
        return y + x + 3
    return g

nf1 = f(1)
nf2 = f(3)

print(nf1(1))
print(nf2(1))
```

We will implement a polynomial "factory" function now. We will start with writing a version which can create polynomials of degree 2.

$$p(x)=a\cdot x^2+b\cdot x+c$$

The Python implementation as a polynomial factory function can be written like this:

```
def polynomial_creator(a, b, c):
    def polynomial(x):
        return a * x ** 2 + b * x + c

    return polynomial


p1 = polynomial_creator(2, 3, -1)
p2 = polynomial_creator(-1, 2, 1)

for x in range(-2, 2, 1):
    print(x, p1(x), p2(x))
```

We can generalize our factory function so that it can work for polynomials of arbitrary degree:

$$\sum_{k=0}^{n} a_k\cdot x^k=a_n\cdot x^n+a_{n-1}\cdot x^{n-1}+...+a_2\cdot x^2+a_1\cdot x+a_0$$

```
def polynomial_creator(*coefficients):
    """ coefficients are in the form a_0, a_1, ... a_n
    """

    def polynomial(x):
```

```
        res = 0
        for index, coeff in enumerate(coefficients):
            res += coeff * x ** index
        return res

    return polynomial

p1 = polynomial_creator(4)
p2 = polynomial_creator(2, 4)
p3 = polynomial_creator(2, 3, -1, 8, 1)
p4 = polynomial_creator(-1, 2, 1)

for x in range(-2, 2, 1):
    print(x, p1(x), p2(x), p3(x), p4(x))
```

Starting with Decorator

Now we have everything ready to define our first simple decorator:

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

def foo(x):
    print("Hi, foo has been called with " + str(x))

print("We call foo before decoration:")
foo("Hi")

print("We now decorate foo with f:")
foo = our_decorator(foo)

print("We call foo after decoration:")
foo(42)
```

Output:

We call foo before decoration:

Hi, foo has been called with Hi

We now decorate foo with f:

We call foo after decoration:

Before calling foo

```
Hi, foo has been called with 42

After calling foo
```

After the decoration "foo = our_decorator(foo)", foo is a reference to the 'function_wrapper'. 'foo' will be called inside of 'function_wrapper', but before and after the call some additional code will be executed, i.e. in our case two print functions.

The decoration in Python is usually not performed in the way we did it in our previous example. We will do a proper decoration now. The decoration occurrs in the line before the function header. The "@" is followed by the decorator function name. We will rewrite now our initial example. Instead of writing the statement.

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

@our_decorator
def foo(x):
    print("Hi, foo has been called with " + str(x))

foo("Hi")
```

We can decorate every other function which takes one parameter with our decorator 'our_decorator'. We demonstrate this in the following. We have slightly changed our function wrapper, so that we can see the result of the function calls:

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        res = func(x)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

@our_decorator
def succ(n):
    return n + 1

succ(10)
```

It is also possible to decorate third party functions, e.g. functions we import from a module. We can't use the Python syntax with the "at" sign in this case:

```
from math import sin, cos
```

```
def our_decorator(func):
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        res = func(x)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

sin = our_decorator(sin)
cos = our_decorator(cos)

for f in [sin, cos]:
    f(3.1415)
```

Summarizing we can say that a decorator in Python is a callable Python object that is used to modify a function, method or class definition. The original object, the one which is going to be modified, is passed to a decorator as an argument. The decorator returns a modified object, e.g. a modified function, which is bound to the name used in the definition.

The previous function_wrapper works only for functions with exactly one parameter. We provide a generalized version of the function_wrapper, which accepts functions with arbitrary parameters in the following example:

```
from random import random, randint, choice

def our_decorator(func):
    def function_wrapper(*args, **kwargs):
        print("Before calling " + func.__name__)
        res = func(*args, **kwargs)
        print(res)
        print("After calling " + func.__name__)
    return function_wrapper

random = our_decorator(random)
randint = our_decorator(randint)
choice = our_decorator(choice)

random()
randint(3, 8)
choice([4, 5, 6])
```

Output:

Before calling random

```
0.40589907513067236
After calling random
Before calling randint
7
After calling randint
Before calling choice
4
After calling choice
```