Functions and Modules in Python

Course Contents

Functions

- Function Parameters
- Default Argument Values
- Keyword Arguments
- The return statement
- Lambda, Map and Filter functions

Modules

- Introduction
- The **import** Methods
- Making Your Own Modules
- The **dir** function

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.

```
say_hello('Mickey','Mouse')
```

```
Hello Mickey Mouse
```

- The function concept is probably the most important building block of any nontrivial software (in any programming language)
- 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.

```
def say_hello(Name, Sname):
    print('Hello', Name, Sname)
```

Calling Functions

- The syntax for calling the function is given below,
  user_defined_function_name(list_of_arguments)

```
def f1():
    print("Hello")
    print('This is first pgm.')
```

```
f1()
f1()
f1()
```

```
Hello
This is first pgm.
Hello
This is first pgm.
Hello
This is first pgm.
```

```
for i in range(5):
      f1()
```

```
Hello
This is first pgm.
Hello
This is first pgm.
Hello
This is first pgm.
Hello
This is first pgm.
Hello
This is first pgm.
```

Function Parameters

- A function can take parameters, which are values you supply to the function so that the function can do something utilizing 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.

Function Parameters

```
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')
```

```
# directly pass literal values
print_max(3, 4)
```

```
4 is maximum
```

```
x = 5
y = 7
```

```
# pass variables as arguments
print_max(x, y)
```

```
7 is maximum
```

Passing Arguments to Function

- In python there are four types of arguments.

    1. Required Arguments
    2. Default Arguments
    3. Keyword Arguments
    4. Variable Length Arguments

Required Arguments

- The required arguments are values passed from *calling* function to *called* function.
- The user defined function can also have no arguments.
- In Required arguments , the number of arguments should be matched in function call and function definition. Otherwise it yields an error.

```
def f1(x):
        print ('The value of x is ', x)

n = input('Enter the value of argument: ')
f1(n)
```

```
Enter the value of argument: 10
The value of x is  10
```

```
def f2(x,y):
        print ('The value of x,y is ', x,y)

n = input('Enter the value of argument: ')
f2(n)
```

```
Enter the value of argument: 12

TypeError: f2() missing 1 required positional
           argument: 'y'
```

Default Argument Values

- For some functions, we 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.
- Note that the default argument value should be a constant.
- Only those parameters which are at the end of the parameter list can be given default argument values i.e. you cannot have a parameter with a default argument value preceding a parameter without a default argument value in the function's parameter list.
- This is because the values are assigned to the parameters by position.
- For example, def func(a, b=5) is valid, but def func(a=5, b) is not valid.

Default Argument Values

```
def say(message, times=3):
    print (message * times)
```

```
say('Hello ')
```

```
Hello Hello Hello
```

```
say('World ', 5)
```

```
World World World World World
```

```
def say1(message='Hi ', times=3):
    print (message * times)
```

```
say1('Hello ')
```

```
Hello Hello Hello
```

```
say1('World ', 5)
```

```
World World World World World
```

```
say1()
```

```
Hi Hi Hi
```

```
def say2(message='Hi ', times):
    print (message * times)
```

```
SyntaxError: non-default argument
             follows default argument
```

Keyword Arguments

- If we have some functions with many parameters and we 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.

Keyword Arguments

```
def add(x,y):
    s = x + y
    print ('x value is', x)
    print ('y value is', y)
    print ('Sum is', s)
```

```
add( 100 , 200)
```

```
x value is 100
y value is 200
Sum is 300
```

```
n1 = int(input("Enter the first number "))
n2 = int(input("Enter the second number "))
```

```
Enter the first number 10
Enter the second number 20
```

```
add( n1,n2)
```

```
x value is 10
y value is 20
Sum is 30
```

```
add( x = n1 , y = n2)
```

```
x value is 10
y value is 20
Sum is 30
```

```
add( y = n1 , x = n2)
```

```
x value is 20
y value is 10
Sum is 30
```

VarArgs parameters

- Sometimes we 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.

```
def total(initial=5, *numbers, **keywords):
```

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

VarArgs parameters

```
def total(initial=5, *numbers, **keywords):
    count = initial
    print ("Initial  :",count)
    print ("Numbers  :",numbers)
    for n in numbers:
        count = count + n
    print ("Keywords :",keywords)
    for key in keywords:
        count = count + keywords[key]
    print ("Count    :",count)
```

```
total(10,21, 32, 3, vegetables=50, fruits=100)
```

```
Initial  : 10
Numbers  : (21, 32, 3)
Keywords : {'vegetables': 50, 'fruits': 100}
Count    : 216
```

```
total(5,1,12,3,54,5,vegetables=50,fruits=100,bags = 10)
```

```
Initial  : 5
Numbers  : (1, 12, 3, 54, 5)
Keywords : {'vegetables': 50, 'fruits': 100, 'bags': 10}
Count    : 240
```

```
total(vegetables=50, fruits=100,bags = 10)
```

```
Initial  : 5
Numbers  : ()
Keywords : {'vegetables': 50, 'fruits': 100, 'bags': 10}
Count    : 165
```

```
total()
```

```
Initial  : 5
Numbers  : ()
Keywords : {}
Count    : 5
```

The return statement

- The return statement is used to return from a function i.e. break out of the function.
- We can optionally return a value from the function as well.
- Note that a return statement without a value is equivalent to return None .
- None is a special type in Python that represents nothingness.
- For example, it is used to indicate that a variable has no value if it has a value of None .
- Every function implicitly contains a return None statement at the end unless you have written your own return statement.
- Multiple values also can be return by using tuple unpacking methodology.

The return statement

```
def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return 'The numbers are equal'
    else:
        return y
```

```
c = maximum(2, 3)

print (c)
print (type(c))
```

```
3
<class 'int'>
```

```
def add(x,y):
    s = x + y
    x = x + 1
    y = y + 1
    return(s,x,y)
```

```
n1 = int(input("Enter the first number "))
n2 = int(input("Enter the second number "))
print("The two input values are ",n1 , n2)
```

```
Enter the first number 10
Enter the second number 20
The two input values are  10 20
```

```
s1, x1 , y1 = add(n1,n2)
print("The sum of two numbers is ",s1)
print("The incremented X value ",x1)
print("The incremented Y value ",y1)
```

```
The sum of two numbers is  30
The incremented X value  11
The incremented Y value  21
```

lambda, map and filter functions

- ***lambda*** operator or ***lambda*** function is used for creating small, one-time and anonymous function objects in Python.

  lambda arguments : expression

- lambda operator can have any number of arguments, but it can have only one expression. It cannot contain any statements and it returns a function object which can be assigned to any variable.

```
def add(x, y):
    return x+y
```

```
print (add(2, 3))
```

```
5
```

```
add = lambda x, y : x + y
print(type(add))

print (add(2, 3))
```

```
<class 'function'>
5
```

lambda, map and filter functions

- ***map*** functions expects a function object and any number of iterables like list, dictionary, etc. It executes the function_object for each element in the sequence and returns a list of the elements modified by the function object.

map(function_object, iterable1, iterable2,...)

```
def multiply2(x):
    return x * 2
```

```
res = multiply2(2)
print(res)
```

```
4
```

```
x = map(multiply2, [1, 2, 3, 4])
x
```

```
<map at 0x1e4bae57358>
```

```
print(type(x))
```

```
<class 'map'>
```

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

```
[2, 4, 6, 8]
```

lambda, map and filter functions

- Using map with lambda

```
x = map(lambda x : x*2, [1, 2, 3, 4])
print(list(x))
```

```
[2, 4, 6, 8]
```

- Multiple iterables to the map function

```
list_1 = [1, 2, 3]
list_2 = [10, 20, 30]

list(map(lambda x, y: x + y, list_1, list_2))
```

```
[11, 22, 33]
```

- Iterating over a dictionary using map and lambda

```
list_a = [{'name': 'python', 'points': 10},
          {'name': 'java', 'points': 8}]

print(list_a)
```

```
[{'name': 'python', 'points': 10},

 {'name': 'java', 'points': 8}]
```

```
list(map(lambda x : x['name'], list_a))
```

```
['python', 'java']
```

```
list(map(lambda x : x['points']*10,  list_a))
```

```
[100, 80]
```

```
list(map(lambda x : x['name'] == "python", list_a))
```

```
[True, False]
```

lambda, map and filter functions

- ***filter*** function expects two arguments, function_object and an iterable.
- function_object returns a boolean value.
- function_object is called for each element of the iterable and filter returns only those element for which the function_object returns true.
- Like map function, filter function also returns a list of element.
- Unlike map function filter function can only have one iterable as input.

filter(function_object, iterable)

lambda, map and filter functions

- Filter function

```
# Even number using filter function
a = [1, 2, 3, 4, 5, 6]
```

```
list(map(lambda x : x % 2 == 0, a))
```

```
[False, True, False, True, False, True]
```

```
list(filter(lambda x : x % 2 == 0, a))
```

```
[2, 4, 6]
```

- Filtering a dictionary using filter and lambda

```
list_a = [{'name': 'python', 'points': 10},
          {'name': 'java', 'poins': 8}]

print(list_a)
```

```
[{'name': 'python', 'points': 10}, {'name': 'java', 'poins': 8}]
```

```
# Filter list of dicts

list(filter(lambda x : x['name'] == 'python', list_a))
```

```
[{'name': 'python', 'points': 10}]
```

Recursive Functions

- Recursion is the process of defining something in terms of itself.
- A function which calls itself is called as a recursive function.
- Through **Recursion** one can Solve problems in easy way while its iterative solution is very big and complex.

```
def decrement(a):
    if (a > 0) :
        print (a)
        decrement(a-1)
    else:
        return()
n = int(input('Enter the value : '))
decrement(n)
```

```
Enter the value : 5
5
4
3
2
1
```

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.
- 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

Local Variables

```
def func():
    x5 = 2
    print ('Changed local x5 to', x5)

func()
print ('Local x5 value is ', x5)
```

```
Changed local x5 to 2
NameError: name 'x5' is not defined
```

```
x = 50

def func():
    print ('x is', x)

func()
print ('x is still', x)
```

```
x is 50
x is still 50
```

The global statement

The basic rules for global keyword in Python are:

- When we create a variable inside a function, it is local by default.
- When we define a variable outside of a function, it is global by default. You don't have to use global keyword.
- We use global keyword to read and write a global variable inside a function.
- Use of global keyword outside a function has no effect.

The global statement

```
x = 50

def func():
    x = 2
    print ('Changed local x to', x)

func()
print ('x is still', x)
```

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

```
x = 50

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

```
func()
print ('x is still', x)
```

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

```
x = 3
print ('Value of x is', x)
```

```
Value of x is 3
```

DocStrings

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

- **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.

```
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)
```

```
5 is maximum
```

```
print (print_max.__doc__)
```

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

Modules

- We have seen how we can reuse code in our program by defining functions once.
- What if we wanted to reuse a number of functions in other programs that we write?
- The answer is **Modules**.
- There are various methods of writing modules, but the simplest way is to create a file with a .py extension that contains functions and variables.
- A module can be imported by another program to make use of its functionality. This is how we can use the Python standard library as well.

import Modules

- First, we will see how to use the standard library modules.

```
import math
print ("Square root of 16 is", math.sqrt(16))
```

```
Square root of 16 is 4.0
```

- If you want to directly import the **sqrt** function into your program (to avoid typing the **math.** every time for it), then we can use the **from … import …** statement.
- In general, we should avoid using this statement and use the import statement instead since our program will avoid name clashes and will be more readable.

```
from math import sqrt
print ("Square root of 16 is", sqrt(16))
```

```
Square root of 16 is 4.0
```

import Modules

- More than one element of a module can be imported using from .. import .. statement.

```
from math import (sqrt , gcd)
print ("Square root of 16 is", sqrt(16))
print ("GCD of 16, 4 is", gcd(16,4))
```

```
Square root of 16 is 4.0
GCD of 16, 4 is 4
```

- All the elements of a module can be imported using from .. import * statement

```
from math import *
print ("Square root of 16 is", sqrt(16))
print ("GCD of 16, 4 is", gcd(16,4))
```

```
Square root of 16 is 4.0
GCD of 16, 4 is 4
```

Making Your Own Modules

- Creating your own modules is easy, we've been doing it all along!
- This is because every Python program is also a module.
- One just have to make sure it has a .py extension.

File Name : **sum_module.py**

```
v1 = 100

def s(x, y):
    return (x + y)
```

```
import sum_module
print(sum_module.s(100,200))
print(sum_module.v1)
```

```
300
100
```

```
from sum_module import (s, v1)
print(s(100,200))
print(v1)
```

```
300
100
```

```
from sum_module import *
print(s(100,200))
print(v1)
```

```
300
100
```

The **dir** function

- We can use the built-in dir function to list the identifiers that an object defines.
- For a module, the identifiers include the functions, classes and variables defined in that module.
- When we supply a module name to the **dir()** function, it returns the list of the names defined in that module.

```
import math
dir(math)
```

```
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'e',
 'erf',
```

```
help(math.sqrt)
```

```
Help on built-in function sqrt in module math:
sqrt(...)
    sqrt(x)

    Return the square root of x.
```