Functions

Defining a Function

- A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.
- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code 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( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

Functions

- A function has three important parts:
- **Name.** Every function has a name that identifies the code to be executed. Function names follow the same rules as variable names; a function name is another example of an identifier
- **Parameters**. A function must be called with a certain number of parameters, and each parameter must be the correct type.
- **Result type**. A function returns a value to its caller. Generally a function will compute a result and return the value of the result to the client. The client's use of this result must be compatible with the function's specified result type. A function's result type and its parameter types can be completely unrelated.

Functions

- Some functions, like print and range, permit clients to pass a variable number of arguments, but most functions, like sqrt, specify an exact number.
- If a client attempts to call a function with too many or too few parameters, the interpreter will issue an error message and refuse to run the program.
- Some functions do not accept any parameters; for example, the function to generate a pseudorandom floating-point number, random, requires no arguments:
  - Random() may display 0.9595266948278349

Functions,Procedures

```
def name(arg1, arg2, ...):
    """documentation"""    # optional doc string
    statements

return                     # from procedure
return expression          # from function
```

Example Function

```
def gcd(a, b):
    "greatest common divisor"
    while a != 0:
        a, b = b%a, a    # parallel assignment
    return b


>>> gcd.__doc__
'greatest common divisor'
>>> gcd(12, 20)
4
```

Defining functions

```
def fib(n):
  """Print a Fibonacci series up to n."""
  a, b = 0, 1
  while b < n:
    print b,
    a, b = b, a+b

>>> fib(2000)
```

- First line is *docstring*
- first look for variables in local, then global
- need global to assign global variables

Returning multiple values

- Python also has the ability to return multiple values from a function call, something missing from many other languages. In this case the return values should be a comma-separated list of values and Python then constructs a *tuple* and returns this to the caller,
- e.g.
- An alternate syntax when dealing with multiple return values is to have Python "unwrap" the tuple into the variables directly by specifying the same number of variables on the left-hand side of the assignment as there are returned from the function, e.g.

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 −

```
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]]
```

Pass by reference vs value

- 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]
```

Function Arguments

- You can call a function by using the following types of formal arguments:
  - Required arguments
  - Keyword arguments
  - Default arguments
  - Variable-length arguments

Required arguments

- Required arguments are the arguments passed to a function in correct positional order.
- Here, the number of arguments in the function call should match exactly with the function definition.

Keyword arguments

- Keyword arguments are related to the function calls.
- When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.
- You can make keyword calls to the *printme()* function in the following ways −

```
def printme( str ):
   "This prints a passed string into this function"
   print(str)
   return

# Now you can call printme function
printme( str = "My string")
```

Keyword arguments

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

  My string

- The following example gives more clear picture. Note that the order of parameters does not matter.

```
def printinfo( name, age ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;
```

- # Now you can call printinfo function

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

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

```
Name:  miki
Age  50
```

Default arguments

- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

```
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;
```

- # Now you can call printinfo function

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

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

```
Name:  miki
Age  50
Name:  miki
Age  35
```

Default Parameters

- We can define our own functions that accept a varying number of parameters by using a technique known as default parameters. Consider the following function that counts down:
- def countdown(n=10):

        for count in range(n, -1, -1):

        print(count)

would print from 10 to 0

But the invocation

        countdown(5)

would print from 5 to 1

Default Parameters

- Non-default and default parameters may be mixed in the parameter lists of function declarations.
- But all default parameters within the parameter list must appear after all the non-default parameters.
- The following are valid:

  - def sum_range(n, m=100):
  - **def sum_range(n=0, m=100):**
  - **def sum_range(n=0, m): is illegal**

Default Parameters

Example:

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

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

Output:

- 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

Variable-length arguments

- You may need to process a function for more arguments than you specified while defining the function.
- These arguments are called *variable-length* arguments and **are not named** in the function definition, unlike required and default arguments.
- Syntax for a function with non-keyword variable arguments is this –

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

- An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments.
- This tuple remains empty if no additional arguments are specified during the function call.

Variable-length arguments

- Following is a simple example −

- # Function definition is here

```
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print "Output is: "
   print arg1
   for var in vartuple:
      print var
   return;
```

Varargs Parameters

- # Now you can call printinfo function
  printinfo(10)
  printinfo(70, 60, 50)
- When the above code is executed, it produces the following result –
- Output is:

  10
- Output is:
  70
  60
  50

Varargs Parameters

- Sometimes you might want to define a function that can take any number of parameters, numbers or keywords, this can be achieved by using the stars :
- def total(initial=5, *numbers, **keywords):
- count = initial
- for number in numbers:
- count += number
- for key in keywords:
- count += keywords[key]
- return count

- print(total(10, 1, 2, 3, vegetables=50, fruits=100))
- Output is 166

The *Anonymous* Functions

- These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword.
- You can use the *lambda* keyword to create small anonymous functions.
- Lambda forms can take any number of arguments but return just one value in the form of an expression.
- The **lambda** keyword in **Python** provides a shortcut for declaring small anonymous functions. **Lambda** functions behave just like regular functions declared with the def keyword. They can be **used** whenever function objects are required.
- They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

lambda functions

- Syntax
- The syntax of *lambda* functions contains only a single statement, which is as follows −

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

- Following is the example to show how *lambda* form of function works −

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;
 # Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

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

```
Value of total :  30
Value of total :  40
```

lambda functions

- # Program to show the use of lambda functions
- double = lambda x: x * 2
- # Output: 10
- print(double(5))
- In the above program, lambda x: x * 2, is the lambda function.
- this function has no name. It returns a function object which is assigned to the identifier double
- double = lambda x: x * 2 is nearly the same as
- Def double(x):
  - Return x* 2

Use of Lambda Function in python

- We use lambda functions when we require a nameless function for a short period of time.
- Lambda functions are used along with built-in functions like filter(), map() etc.
- The filter function in Python takes in a function and a list as arguments.
- The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True

Use of Lambda Function in python

- function to filter out only even numbers from a list.
- Program to filter out only the even items from a list
- my_list = [1, 5, 4, 6, 8, 11, 3, 12]
- new_list = list(filter(lambda x: (x%2 == 0) , my_list))
- # Output: [4, 6, 8, 12]
- print(new_list)

Use of Lambda Function in python

- The map function in Python takes in a function and a list.
- The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.
- # Program to double each item in a list using map()
- my_list = [1, 5, 4, 6, 8, 11, 3, 12]
- new_list = list(map(lambda x: x * 2 , my_list))
- # Output: [2, 10, 8, 12, 16, 22, 6, 24]
- print(new_list)

Use of Lambda Function in python

- # Python Program to display the powers of 2 using anonymous function
- # Change this value for a different result
- terms = 10
- # Uncomment to take number of terms from user
- #terms = int(input("How many terms? "))
- # use anonymous function
- result = list(map(lambda x: 2 ** x, range(terms)))
- # display the result
- print("The total terms is:",terms)
- for i in range(terms):
  - print("2 raised to power",i,"is",result[i])

Recursion

```
def factorial(n):
    if n==0:
        return 1
    else
        return n * factorial(n-1)

def main():
        print(“6!= “, factorial(6))
        print(“4!= “, factorial(6))

main()
```