Functions:

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.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these 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]
```

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

Example:

Here is the simplest form of a Python function. This function takes a string as input parameter and prints it on standard screen.

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

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function:

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str;
   return;
# Now you can call printme function
printme("I'm first call to user defined function!");
printme("Again second call to the same function");
```

When the above code is executed, it produces the following result:

```
I'm first call to user defined function!
Again second call to the same function
```

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:

```
#!/usr/bin/python

# 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.

```
#!/usr/bin/python

# 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.

To call the function *printme()*, you definitely need to pass one argument, otherwise it would give a syntax error as follows:

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str;
   return;

# Now you can call printme function
printme();
```

When the above code is executed, it produces the following result:

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    printme();
TypeError: printme() takes exactly 1 argument (0 given)
```

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 also make keyword calls to the *printme()* function in the following ways:

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str;
   return;

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

When the above code is executed, it produces the following result:

```
My string
```

Following example gives more clear picture. Note, here order of the parameter does not matter.

```
#!/usr/bin/python

# Function definition is here
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. Following example gives an idea on default arguments, it would print default age if it is not passed:

```
#!/usr/bin/python

# Function definition is here
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
```

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.

The general 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 will hold the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example:

```
#!/usr/bin/python
# 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;
# 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
```

The *Anonymous* Functions:

You can use the *lambda* keyword to create small anonymous functions. These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. 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.

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:

```
#!/usr/bin/python

# 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
```

The *return* Statement:

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.

All the above examples are not returning any value, but if you like you can return a value from a function as follows:

```
#!/usr/bin/python

# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2
   print "Inside the function : ", total
   return total;

# Now you can call sum function
total = sum( 10, 20 );
print "Outside the function : ", total
```

When the above code is executed, it produces the following result:

```
Inside the function :  30
Outside the function :  30
```

Scope of Variables:

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python:

- Global variables
- Local variables

Global vs. Local variables:

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example:

```
#!/usr/bin/python

total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2; # Here total is local variable.
   print "Inside the function local total : ", total
   return total;

# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total : ", total
```

When the above code is executed, it produces the following result:

```
Inside the function local total :  30
Outside the function global total :  0
```

Modules:

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Example:

The Python code for a module named *aname* normally resides in a file named *aname.py*. Here's an example of a simple module, support.py

```
def print_func( par ):
   print "Hello : ", par
   return
```

The *import* Statement:

You can use any Python source file as a module by executing an import statement in some other Python source file. The *import* has the following syntax:

import module1[, module2[,... moduleN]

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module hello.py, you need to put the following command at the top of the script:

```
#!/usr/bin/python

# Import module support
import support

# Now you can call defined function that module as follows
support.print_func("Zara")
```

When the above code is executed, it produces the following result:

```
Hello : Zara
```

A module is loaded only once, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur.

The *from...import* Statement

Python's *from* statement lets you import specific attributes from a module into the current namespace. The *from...import* has the following syntax:

from modname import name1[, name2[, ... nameN]]

For example, to import the function fibonacci from the module fib, use the following statement:

from fib import fibonacci

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

The *from...import* * Statement:

It is also possible to import all names from a module into the current namespace by using the following import statement:

from modname import *

This provides an easy way to import all the items from a module into the current namespace; however, this statement should be used sparingly.

Locating Modules:

When you import a module, the Python interpreter searches for the module in the following sequences:

- The current directory.
- If the module isn't found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the **sys.path** variable. The sys.path variable contains the current directory, PYTHONPATH, and the installation-dependent default.

The *PYTHONPATH* Variable:

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system:

```
set PYTHONPATH=c:\python20\lib;
```

And here is a typical PYTHONPATH from a UNIX system:

```
set PYTHONPATH=/usr/local/lib/python
```

Namespaces and Scoping:

Variables are names (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.

Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.

Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.

The statement *global VarName* tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

For example, we define a variable *Money* in the global namespace. Within the function *Money*, we assign *Money* a value, therefore Python assumes *Money* as a local variable. However, we accessed the value of the local variable *Money* before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

```
#!/usr/bin/python

Money = 2000
```

```
def AddMoney():
   # Uncomment the following line to fix the code:
   # global Money
   Money = Money + 1

print Money
AddMoney()
print Money
```

The dir( ) Function:

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example:

```
#!/usr/bin/python

# Import built-in module math
import math

content = dir(math)

print content;
```

When the above code is executed, it produces the following result:

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

Here, the special string variable *__name__* is the module's name, and *__file__* is the filename from which the module was loaded.

The *globals()* and *locals()* Functions:

The *globals()* and *locals()* functions can be used to return the names in the global and local namespaces depending on the location from where they are called.

If locals() is called from within a function, it will return all the names that can be accessed locally from that function.

If globals() is called from within a function, it will return all the names that can be accessed globally from that function.

The return type of both these functions is dictionary. Therefore, names can be extracted using the keys() function.

The *reload()* Function:

When the module is imported into a script, the code in the top-level portion of a module is executed only once.

Therefore, if you want to reexecute the top-level code in a module, you can use the *reload()* function. The reload() function imports a previously imported module again. The syntax of the reload() function is this:

reload(module_name)

Here, *module_name* is the name of the module you want to reload and not the string containing the module name. For example, to reload *hello* module, do the following:

reload(hello)

Packages:

- pip is the preferred installer program. Starting with Python 3.4, it is included by default with the Python binary installers.
- A *virtual environment* is a semi-isolated Python environment that allows packages to be installed for use by a particular application, rather than being installed system wide.
- venv is the standard tool for creating virtual environments, and has been part of Python since Python 3.3. Starting with Python 3.4, it defaults to installing pip into all created virtual environments.
- virtualenv is a third party alternative (and predecessor) to venv. It allows virtual environments to be used on versions of Python prior to 3.4, which either don't provide venv at all, or aren't able to automatically install pip into created environments.
- The Python Packaging Index is a public repository of open source licensed packages made available for use by other Python users.
- the Python Packaging Authority are the group of developers and documentation authors responsible for the maintenance and evolution of the standard packaging tools and the associated metadata and file format standards. They maintain a variety of tools, documentation, and issue trackers on both GitHub and BitBucket.
- distutils is the original build and distribution system first added to the Python standard library in 1998. While direct use of distutils is being phased out, it still laid the foundation for the current packaging and distribution infrastructure, and it not only remains part of the standard library, but its name lives on in other ways (such as the name of the mailing list used to coordinate Python packaging standards development).

Deprecated since version 3.6: pyvenv was the recommended tool for creating virtual environments for Python 3.3 and 3.4, and is deprecated in Python 3.6.

Changed in version 3.5: The use of venv is now recommended for creating virtual environments.

Basic usage

The standard packaging tools are all designed to be used from the command line.

The following command will install the latest version of a module and its dependencies from the Python Packaging Index:

```
python -m pip install SomePackage
```

Note

For POSIX users (including Mac OS X and Linux users), the examples in this guide assume the use of a virtual environment.

For Windows users, the examples in this guide assume that the option to adjust the system PATH environment variable was selected when installing Python.

It's also possible to specify an exact or minimum version directly on the command line. When using comparator operators such as >, < or some other special character which get interpreted by shell, the package name and the version should be enclosed within double quotes:

```
python -m pip install SomePackage==1.0.4    # specific version
python -m pip install "SomePackage>=1.0.4"  # minimum version
```

Normally, if a suitable module is already installed, attempting to install it again will have no effect. Upgrading existing modules must be requested explicitly:

```
python -m pip install --upgrade SomePackage
```

More information and resources regarding pip and its capabilities can be found in the Python Packaging User Guide.

Creation of virtual environments is done through the venv module. Installing packages into an active virtual environment uses the commands shown above.

How do I ...?

These are quick answers or links for some common tasks.

... install pip in versions of Python prior to Python 3.4?

Python only started bundling pip with Python 3.4. For earlier versions, pip needs to be "bootstrapped" as described in the Python Packaging User Guide.

... install packages just for the current user?

Passing the --user option to python -m pip install will install a package just for the current user, rather than for all users of the system.

… install scientific Python packages?

A number of scientific Python packages have complex binary dependencies, and aren't currently easy to install using pip directly. At this point in time, it will often be easier for users to install these packages by other means rather than attempting to install them with pip.

… work with multiple versions of Python installed in parallel?¶

On Linux, Mac OS X, and other POSIX systems, use the versioned Python commands in combination with the -m switch to run the appropriate copy ofpip:

```
python2   -m pip install SomePackage  # default Python 2
python2.7 -m pip install SomePackage  # specifically Python 2.7
python3   -m pip install SomePackage  # default Python 3
python3.4 -m pip install SomePackage  # specifically Python 3.4
```

Appropriately versioned pip commands may also be available.

On Windows, use the py Python launcher in combination with the -m switch:

```
py -2   -m pip install SomePackage  # default Python 2
py -2.7 -m pip install SomePackage  # specifically Python 2.7
py -3   -m pip install SomePackage  # default Python 3
py -3.4 -m pip install SomePackage  # specifically Python 3.4
```