Functions and Modules in Python

Functions

Functions are the block of code that perform a functionality. it has certain advantages like

- It reduce its complexity to some degree and
- it creates a number of well defined, documented boundaries within the program.

Syntax

```
def <function-name>(value-to-be-passed-to-argument):
    s1
    s2
    s3
    .
    .
```

You can create functions in a program that:

- can have arguments, if needed
- can perform certain functionality
- can return a result

for example

```
def calSomething(x):
    r = 2 * x ** 2
    return r
```

where

- def means a function definition is starting.
- identifier following 'def' is the name of the function, i.e. here the function name is calSomthing.
- the variable/identifier inside the parentheses are the arguments or parameters .
- there is a colon at the end of def line, meaning it require a block.
- the statements intended below the function, define the functionality of function.
- The return statement returns the computed result.

Using a Function[Calling a function]

To use a function that has been defined earlier, you need to write a function call statement in Python.

<function-name>(value-to-be-passed-to-argument)

for e.g.

```
calSomthing(5)          # value 5 is being sent as argument
```

Python function types

Python comes preloaded with many function-definition that you can use as per your need. You can even create new functions. Broadly, Python functions can belong to one of the following three categories:

1. **Built-in functions**
   These are pre-define functions and are always available for use. You have used some of them – len(), type(), int(), input()
2. **Functions defined in modules**
   These functions are pre-defined in a particular module and can only be used when the corresponding module is imported. for e.g. if you want to use predefine function inside the module say sin(), you need to first import the module math.
3. **User define functions**
   These are defined by the programmer. As programmers you can create your own functions.

before continuing function we must first discuss module first then we come on function again.

Modules

As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module.

A Python module is a file (.py file) containing variables, class definitions, statements and functions related to a particular task. **It has following structure:**

doc strings	Triple quoted comments; useful for documentation purpose. For documentation, the docstring should be the first string stored inside a module/function-body/class.
Variable and constant	Labels for data.
Classes	Template/blueprint to create object of a certain type.
Objects	Instances of class.
Statement	Instructions.
Functions	Named group instructions.

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable __name__. The Python module that come preloaded with Python are called **standard library modules**. Two such standard library modules are math and random module.

For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Save the module with fibo.py

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

>>> fibo.__name__
```

```
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>> f = fibo.fib2
>>> f(50)
[1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Importing Modules in a Python Program

Python provides import statement to import modules in a program. The import statement can be used in two forms:

i) to import entire module

```
import <module>
```

ii) to import selected objects from a module

```
from <module> import <object>
```

Import Entire Module

Syntax

```
import module1 [, module2  [,…….
```

The import statement internally executes the code of the module file and then makes it available to your program. That is everything defined inside the module – function definitions variables, constant etc.

After importing a module, you can use any function/definition of the imported module as per following syntax:

```
<module-name>.<function-name>()
```

The name of module is stored inside a constant __name__. You can use it like:

```
>>>import time
>>>print( time.__name__)
time
```

Importing selected Objects from a Module

If you want to import some selected items, not all from a module, then you can use `from <module> import statement.`

To import single Object

```
from math import pi
```

Now, the constant pi can be used and you need not prefix it with module name. That is to print the value of pi, you will write

```
>>>print( pi)
```

Do not use module name with imported object if important through form <module> import command because now the imported object is part of your program's environment.

To import Multiple Objects

If you want to import multiple objects from the module like this so that you don't have to prefix the module's name, you can write the comma separated list of objects after keyword **import.** For instance, to import just two functions sqrt() and pow() from math module,

```
from math import sqrt, pow
```

To import all Objects of a Module

If you want to import all the items from the module like this so that you don't have to prefix the module's name you can write:

```
from <module> import *
from math import *
```

Now you can use all the defined functions, variables etc from the math module, without having to prefix module's name to the imported item name.

Python's Processing of import <module> Command

In Python term namespace can be thought of as a named environment holding logical grouping of related objects. You can think of it as named list of some names.

Processing of import <module> command

When you issue import <module> command, internally following things take place:

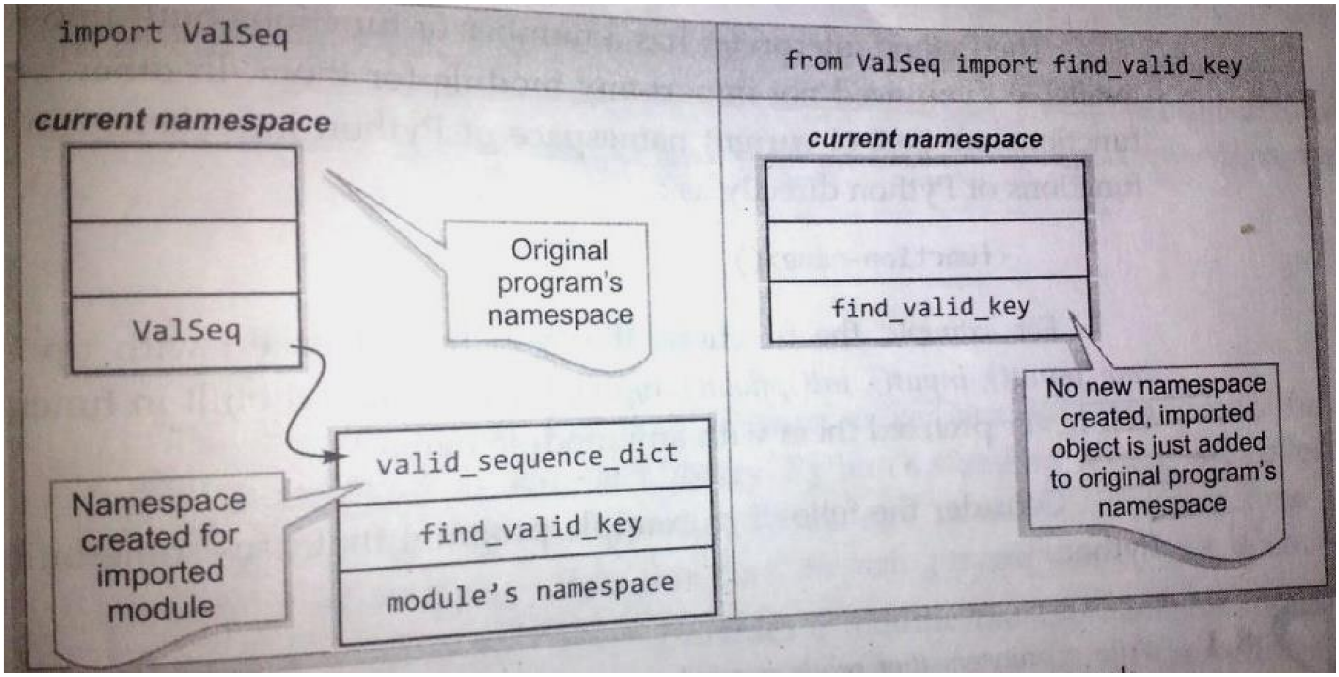
- the code of imported module is interpreted and executed.
- defined functions and variables created in the modules are now available to the program that imported module.
- For imported module, a new namespace is setup with the same name as that of the module.

For instance, you imported module is myMod in your program. Now all the objects of module myMod would be referred as myMod.<object-name>.

Processing of from <module> import <object> command

When you issue from <module> import <object> command, internally following things take palce:

- the code of imported module is interpreted and executed.
- only the asked functions and variables from the module are made available to the program.
- no new namespace is created, the imported definition is just added in the current namespace.



Difference between import<module> and from <module> import command

If your program already has a variable with the same name as the one imported via module, then the variable in your program will hide imported member with same name because there cannot be two variables with the same name in one namespace.

Invoking Python's Built-in Functions

The Python interpreter has a number of functions built into it that are always available; you need not import any module for them. In other words, the built in functions are part of current namespace of Python interpreter. So you use built-in functions of Python directly as:

<function-name>()

for e.g. int(), len(), type(), float(), print(), input()

The Module Search Path

When a module named spam is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named spam.py in a list of directories given by the variable sys.path. sys.path is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
- The installation-dependent default.

if we create our own module say fibo.py in the above example we need to save it in either lib directory or if we want to save it in another directory then set the path of that directory to PYTHONPATH variable.

for e.g. we save the fibo.py in the directory e:\myModules\fibo.py then set PYTHONPATH to e:\myModules\ then it will be interpret.

```
>>> import fibo
>>> dir(fibo)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'fib', 'fib2']
```

The variable sys.path is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the environment variable PYTHONPATH, or from a built-in default if PYTHONPATH is not set.

```
>>> import sys
>>> sys.path
['', 'C:\\Program Files (x86)\\Python\\Lib\\idlelib', 'C:\\Program Files (x86)\\Python\\include', 'E:\\myModules', 'C:\\Program Files (x86)\\Python', 'C:\\Program Files (x86)\\Python\\python36.zip', 'C:\\Program Files (x86)\\Python\\DLLs', 'C:\\Program Files (x86)\\Python\\lib', 'C:\\Program Files (x86)\\Python\\lib\\site-packages']
```

You can modify it using standard list operations:

```
sys.path.append('E:\myMoreModules\')
```

The dir() Function

The built-in function dir() is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
```

```
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'argv',
 'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook',
 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
 'thread_info', 'version', 'version_info', 'warnoptions']
```

Note that it lists all types of names: variables, modules, functions, etc.

Packages

Packages are collection of modules. Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name A.B designates a sub module named B in a package named A.

Suppose you want to design a collection of modules (a "package") for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: .wav, .aiff, .au), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package:

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format
conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

The __init__.py files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as string, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, __init__.py can just be an empty file, but it can also execute initialization code for the package or set the __all__ variable.

Users of the package can import individual modules from the package, for example:

```
import sound.effects.echo
```

This loads the submodule sound.effects.echo. It must be referenced with its full name.

```
sound.effects.echo.echofilter(input, output, delay=0.7,
atten=4)
```

An alternative way of importing the sub module is:

```
from sound.effects import echo
```

This also loads the sub module echo, and makes it available without its package prefix, so it can be used as follows

```
echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```
from sound.effects.eco import echofilter
```

Again, this loads the sub module echo, but this makes its function echofilter() directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```

Note that when using from package import item, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The import statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an ImportError exception is raised.

Importing * From a Package

Now what happens when the user writes from sound.effects import *? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The import statement uses the following convention: if a package's __init__.py code defines a list named __all__, it is taken to be the list of module names that should be imported when from package import * is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing * from their package. For example, the file sound/effects/__init__.py could contain the following code:

__all__ = ["echo", "surround", "reverse"]

This would mean that from sound.effects import * would import the **three named submodules** of the sound package.