MODULES

You have seen how you can reuse code in your program by defining functions once. What if you wanted to reuse a number of functions in other programs that you write? As you might have guessed, 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.

Another method is to write the modules in the native language in which the Python interpreter itself was written. For example, you can write modules in the C programming language and when compiled, they can be used from your Python code when using the standard Python interpreter.

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

Example (save as module_using_sys.py):

```
import sys

print('The command line arguments are:')
```

```
for i in sys.argv:
    print i
print '\n\nThe PYTHONPATH is', sys.path, '\n'
```

Output:

```
$ python module_using_sys.py we are arguments
The command line arguments are:
module_using_sys.py
we
are
arguments
The PYTHONPATH is ['/tmp/py',
# many entries here, not shown here
'/Library/Python/2.7/site-packages',
'/usr/local/lib/python2.7/site-packages']
```

How It Works

First, we **import** the `sys` module using the `import` statement. Basically, this translates to us telling Python that we want to use this module.

The sys module contains functionality related to the Python interpreter and its environment i.e. the **sys**tem.

When Python executes the import sys statement, it looks for the sys module. In this case, it is one of the built-in modules, and hence Python knows where to find it.

If it was not a compiled module i.e. a module written in Python, then the Python interpreter will search for it in the directories listed in its sys.path variable. If the module is found, then the statements in the body of that module are run and the module is made **available** for you to use. Note that the initialization is done only the **first** time that we import a module.

The argv variable in the sys module is accessed using the dotted notation i.e. sys.argv. It clearly indicates that this name is part of the sys module. Another advantage of this approach is that the name does not clash with any argvvariable used in your program.

The sys.argv variable is a **list** of strings (lists are explained in detail in a later chapter. Specifically, the sys.argv contains the list of **command line arguments** i.e. the arguments passed to your program using the command line.

If you are using an IDE to write and run these programs, look for a way to specify command line arguments to the program in the menus.

Here, when we execute python module_using_sys.py we are arguments, we run the module module_using_sys.py with the python command and the other things that follow are arguments passed to the program. Python stores the command line arguments in the sys.argv variable for us to use.

Remember, the name of the script running is always the first argument in the sys.argv list. So, in this case we will have'module_using_sys.py' as sys.argv[0], 'we' as sys.argv[1], 'are' as sys.argv[2] a nd 'arguments' assys.argv[3]. Notice that Python starts counting from 0 and not 1.

The sys.path contains the list of directory names where modules are imported from. Observe that the first string in sys.path is empty - this empty string indicates that the current directory is also part of the sys.path which is same as the PYTHONPATH environment variable. This means that you can directly import modules located in the current directory. Otherwise, you will have to place your module in one of the directories listed in sys.path.

Note that the current directory is the directory from which the program is launched. Run import os; print os.getcwd() to find out the current directory of your program.

Source: http://www.swaroopch.com/notes/python/