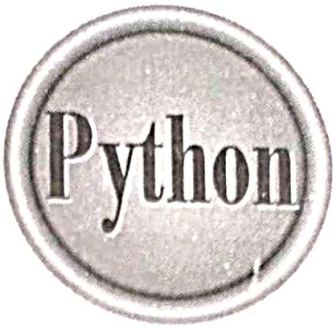
The Complete Reference




# Chapter 2

## Python Fundamentals

Before we start explaining the semantics of the language, you need to know how to execute Python programs. Python is slightly different from many other scripting languages in that there are a number of different ways in which you can execute Python statements, including interactively directly to the Python interpreter.

Many programming languages rely on the following sequence of events for executing an application:

1. Write the application in one or more source files.
2. Compile the source files into object files.
3. Link the object files into an application.
4. Execute the application.

With scripting languages, the steps are simpler because the interpreter works directly with the source file. So the sequence is more likely to be as follows:

1. Write the application in one or more source files.
2. Execute the interpreter, supplying the main source file.

With Perl, Python, and some other scripting languages, steps 2 and 3 from the first list are actually built into the interpreter. When you execute an application, the interpreter compiles the source into an internal bytecode format and then executes the compiled program using a "virtual machine." The virtual machine executes the bytecode in the same way that a compiled application executes native machine code.

## Executing Python Programs

Python follows the same basic process for writing applications as other programming languages do; the Python interpreter takes raw-text source code and executes each statement. The difference in Python is that you can also execute statements directly within the interpreter—you don't have to place Python statements into a file before you execute them.

In addition, Python also allows you to record the compiled bytecode instructions in a file and then execute the bytecode directly, without going through the (relatively) lengthy process of compiling the source code first. Although this technique is not normally used for applications, it is used for modules and extensions to the Python language.

There are also other ways in which you can execute Python applications. Let's take a look at each one individually.

### Interactively

The Python interpreter works slightly differently from other interpreters in that you can enter Python statements directly into the interpreter, rather than having to create

a file and then execute that file. This is very useful when you want to quickly try a particular statement or for very short applications.

## Unix

Under Unix (details on Windows and MacOS follow), you can just execute the Python interpreter at the command line and start typing in Python statements, as in the following example:

```
$ python
Python 2.0 (#4, Dec 16 2000, 07:30:29)
[GCC 2.95.2 19991024 (release)] on sunos5
Type "copyright", "credits" or "license" for more information.
>>>
```

When you enter a Python statement, the interpreter automatically compiles and then executes the statement—you don't have to go through the compilation stage first. The interactive approach means you can try different statements all within the same session. You can modify the statements at the command line before they are executed. Once you press ENTER, the statement is executed—you can't go back and modify it.

If you want to try it, enter the following statement at the Python command line:

```
>>> print 63*56
3528
>>>
```

As you can see, you get the response and the correct result immediately—the statement has been compiled and executed for you on the fly.

The interactive interface is not one shot—you can execute multiline statements just as if you were entering the individual statements into a single file to be executed. This means you can enter very small programs straight into the interpreter to see what happens. Consider the following example:

```
Python 2.0 (#4, Dec 16 2000, 07:30:29)
[GCC 2.95.2 19991024 (release)] on sunos5
Type "copyright", "credits" or "license" for more information.
>>> pi = 3.141592654
>>> radius = 4
>>> area = pi*(radius**2)
>>> print 'Area of circle is:',area
Area of circle is: 18.849555924
>>>
```

Don't worry too much about the syntax for the moment—we'll be using the interactive interface a lot in this book to quickly show the effects and operation of statements and functions within Python. This is precisely what it is there for—to try out statements.

To exit Python's interactive interpreter, press the end-of-file key combination. For most Unix terminals this is CTRL-D.

### Windows

Once you have installed the Python interpreter, you should be able to execute Python statements interactively in exactly the same fashion as you would under Unix:

```
C:\> python
Python 2.0 (#8, Oct 16 2000, 17:27:58) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
>>>
```

You may need to add the directory that contains the Python interpreter to your %PATH%. Under Windows 95/98, you need to add this directory to your **AUTOEXEC.BAT** file. Under Windows NT, you need to modify the PATH setting in the System Control Panel. If you prefer, you can find the command-line Python interpreter in the Start Menu.

Once you are in the interactive interpreter, it works exactly like the Unix version. However, to exit the interpreter you need to use CTRL-Z.

### MacOS

Under MacOS, you need to start the Python application. This opens a simple, terminal-style window, which is the interactive interface to the Python interpreter. Figure 2-1 shows the Python application in action under MacOS.

## From a File

The difficulty with entering statements interactively is that you can never store the statements you have entered—they are executed and essentially deleted immediately. Just like other scripting languages, you can also put Python statements into a file and execute the file as a whole application. Under other scripting languages, this is called a *script file*, a *batch file*, or even an *application*.

With Python, script files are called *modules*. There is nothing special about the format of a Python module; it's a simple text file so you can edit it with your favorite text editor—Emacs, Notepad, BBEdit, whatever you like. The major benefit of using a module is that you don't have to reenter the statements; you can use them in the module again and again.

By tradition, you should give your Python modules the extension .py to indicate that they are Python scripts. Under Unix and MacOS, the significance of the extension given to the file is ignored, but under Windows, the extension enables the module to

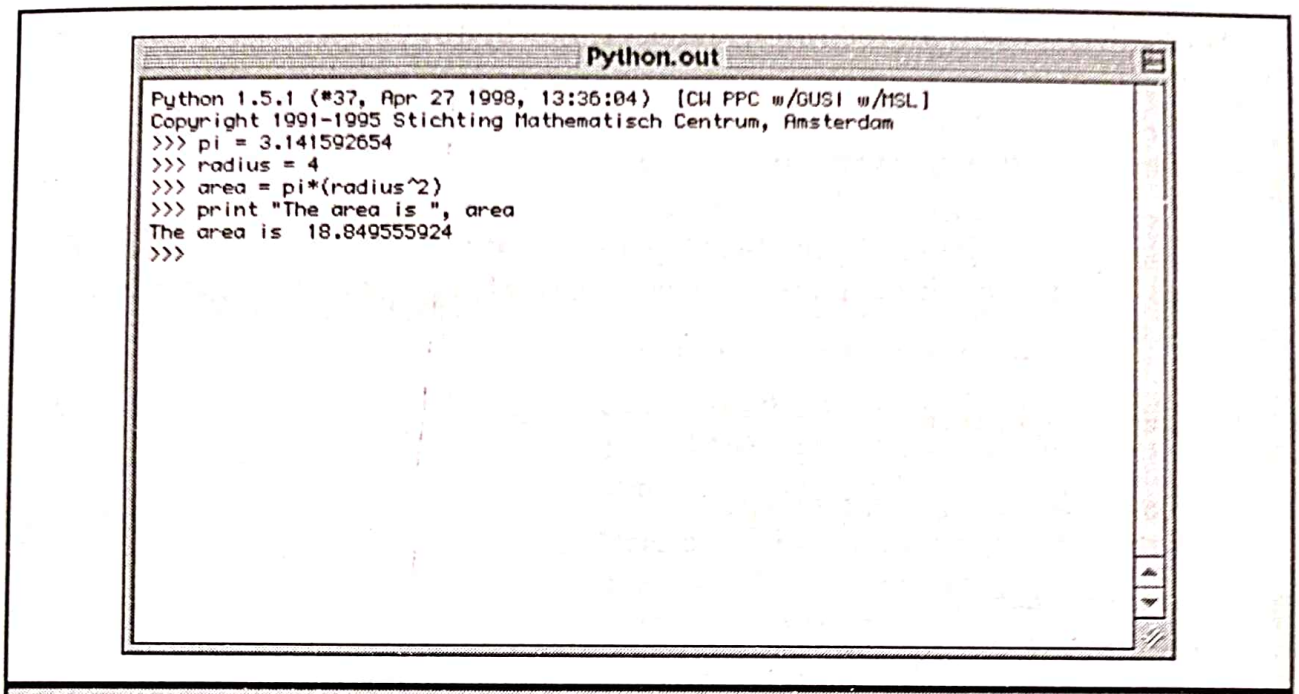



**Figure 2-1.** *Using the Python interpreter under MacOS*

be executed just by double-clicking on the file. We'll explain the different platform-specific options shortly.

There is one advantage with Python modules that is not available in other languages: With Python, any module can be imported by another module without requiring modification—there is no need to modify the original module or change its name. You don't even need to write the module in a special way; just save it as a file with the .py extension. This makes the process of sharing previous functions and objects much easier and promotes much better reuse of the objects you have created before.

## As a Unix Script

When you create a Python module under Unix, to execute the statements in the module you need to supply the name of the module to the Python interpreter. Let's try it with a simple script:

```
import sys
print 'Hello ', sys.argv[1]
```

Save these commands in a file called **test.py**. Then supply the filename as the first argument to the Python interpreter:

```
$ python test.py Martin
Hello Martin
```

Alternatively, insert the following line as the first line of the file. This is known as the *shebang line:*

```
#!/usr/local/bin/python
```

You also need to change the mode of the file so that it is executable; this makes the Unix shell examine the shebang line to determine what application to use to execute the text script:

```
$ chmod 755 test.py
```

You can now run the script directly:

```
$ test.py Martin
Hello Martin
```

The shebang line tells Unix what application to use to execute the file. The shebang line in the preceding example specifies the location of the Python interpreter directly. A more compatible method, which causes the script to search the value of the **PATH** environment variable, is to use the following shebang line instead:

```
#!/usr/bin/env python
```

Make sure you check where the **env** utility is located; otherwise the shell will signal an error. You might find **env** in **/bin**, **/usr/bin**, **/usr/local/bin**, or **/usr/sbin**.

**Configuring the Python Interpreter** Whichever method you decide to use when executing a Python script, there may be times when you want to supply additional command-line switches to the Python interpreter. For scripts to be manually executed by the Python interpreter, you must supply those options before you supply the script name. For example, to cause the interpreter to go into interactive mode after the script has been executed, you need to specify the following:

```
$ python -i sample.py
```

Or, use the following shebang line:

```
#!/usr/local/bin/python -i
```

Table 2-1 contains the full list of command-line options and environment variables.

| Option | Environment Variable | Description |
|---|---|---|
| -d | PYTHONDEBUG | Generates debug information from the interpreter after the script has been compiled. |
| -i | PYTHONINSPECT | Causes the interpreter to go into interactive mode after the script has been executed. |
| -O | PYTHONOPTIMIZE | Optimizes the bytecode generated by the interpreter before it is executed. |
| -OO | | Optimizes the bytecode and removes the embedded document strings from the optimized code before it is executed. |
| -S | | Does not automatically import the **site.py** module, which contains site-specific Python statements, when the interpreter starts. |
| -t | | Generates warnings when tab-based indentation of the script is inconsistent. See Chapter 3 for details on Python blocks. |
| -tt | | Generates errors (and stops parsing) when the indentation of the script using tabs is inconsistent. |
| -u | PYTHONUNBUFFERED | Forces the standard output and error filehandles to operate unbuffered. If not specified, then buffered output is used. |
| -v | PYTHONVERBOSE | Generates information about modules imported by the script when executed. |
| -x | | Skips the first line of the source file. Useful when executing a script on a different platform than the source when you want to skip the shebang line. |
| -X | | Disables the class-based exceptions that are built into the interpreter. (See Chapter 6 for more information.) |
| -c *cmd* | | Uses *cmd* as the script source instead of a source file. |
| - | | Reads the source file from the standard input. |

**Table 2-1.** *Python Command-Line Options and Environment Variables*

Where a variable is available for configuring an option, its existence and value are used to determine whether the option is set. For example, merely creating the variable is not enough, as in the following example:

```
$ export set PYTHONINSPECT
$ sample.py
Hello World!
```

You must give the variable a value:

```
$ export set PYTHONINSPECT=1
$ sample.py
Hello World!
>>>
```

In addition, Python supports the environment variables described in Table 2-2.

## On a Windows Host

Under Windows, you have two options. If you want to execute a Python script from the DOS prompt, just supply the name of the file to the application:

```
C:\> python test.py
```

The other alternative is to define the **.py** extension as a file type within Windows Explorer (the File Types tab under Folder Options). The Python installer does this for you when the interpreter is installed. When you double-click on the file within a

| Variable | Description |
|---|---|
| PYTHONSTARTUP | The name of a file to be executed when starting the interpreter in interactive mode. |
| PYTHONPATH | A list of directories (separated by colons under Unix or semicolons under Windows) to be searched when importing modules. The resulting list is available internally as **sys.path**. |
| PYTHONHOME | The directory in which the core Python libraries can be found. Defaults to **$PYTHONHOME/python2.0**. |

**Table 2-2.** *Python Environment Variables*

Windows Explorer window, the Python interpreter opens with that file displayed, just as if you'd typed the instruction on the command line.

The only problem with this double-click method is that you cannot supply any arguments on the command line to the script; the script is only interactive if you add the necessary code to request information from the user when the script is executed. The solution is to create a batch file that executes the Python interpreter, your script, and any arguments you want to supply to the script.

One final option eliminates the need to open the command-line prompt at all. If you name the file with a .pyw extension, then a DOS command prompt is not opened when the script is run, allowing the script to appear just like a normal Windows application. However, in this mode there is no form of interaction unless it's been defined within the script—you'll need to open a console window or more likely, develop a Tk-based interface within the script to allow interaction from the user. Chapter 14 describes how to use Tk to develop interfaces within Python.

Python under Windows does not support the shebang line that is used under Unix to define which options to be used when running the interpreter. This means that if you want to supply specific options to the interpreter when you run a script, you need to embed the call within a batch file. This is actually a limitation of the operating system, rather than a limitation of Python. All of the command-line options available under Windows are identical to those under Unix. See Table 2-1 for details.

## On a MacOS Host

Under MacOS there is no way to identify the type of file by its name. The MacOS does not use extensions to identify file types; instead, it employs a special system using special four-character type and creator codes. For example, a Python applet under MacOS has a type TEXT and a creator Pyth. Without special tools such as FileTyper or Snitch, there is no way to modify this information.

To execute a Python script on a MacOS machine, you need to create a normal text file. You can do this with the SimpleText application, or you can use an editor such as BBEdit or Emacs to create the file. You can even use Microsoft Word or AppleWorks—just make sure that when you save the file, you save it as a normal text file, not as a Word or AppleWorks document.

Once you create the script file, you need to drag and drop the file onto the Python application in order for the application to execute the script. Double-clicking on the file at this stage only opens the file again within the editor. Note that when the file is executed, the Python application is opened, the script is run, and then the application exits. If you want to display some information to the user, you need to either pause the output by waiting for some input or delay the normal exit procedure of the program. See the next section, "Configuring Python under MacOS," for details on how to handle other execution options.

If you want to create a Python application that can just be double-clicked to be executed, you need to use the **BuildApplet** application. Drag and drop your Python script onto this application and a new application will be created. When you double-click

on the new application, the Python library is loaded by the application and then the script is executed, just as if you had dropped the script onto the Python interpreter.

Note, however, that the application is not completely self-contained—you cannot supply the application to end users without also asking them to install the Python interpreter. The application that is created is really only a placeholder to allow for easier execution of a Python script.

Also, if you modify the source script, you will need to rebuild the applet to incorporate the changes—the modifications are not automatically applied to the applet as well.

**Configuring Python under MacOS** MacOS does not support environment variables or command-line options. To modify the operation of the interpreter or a Python applet, you must use the **EditPythonPrefs** application. If you want to modify the main Python application, and therefore any scripts dragged and dropped onto the interpreter, just double-click on the Python application. The first window, as seen in Figure 2-2, enables you to specify the library search path for the interpreter. You can also specify the HOME location for the execution of the interpreter; this information is used when looking for modules to be imported. Modifying this information is analogous to modifying the **PYTHONPATH** and **PYTHONHOME** environment variables under Windows and Unix.

If you click on Default Startup Options, you can configure many of the options available to Unix and Windows users, as well as some of the options that are specific to the MacOS platform. Figure 2-3 contains this option window.

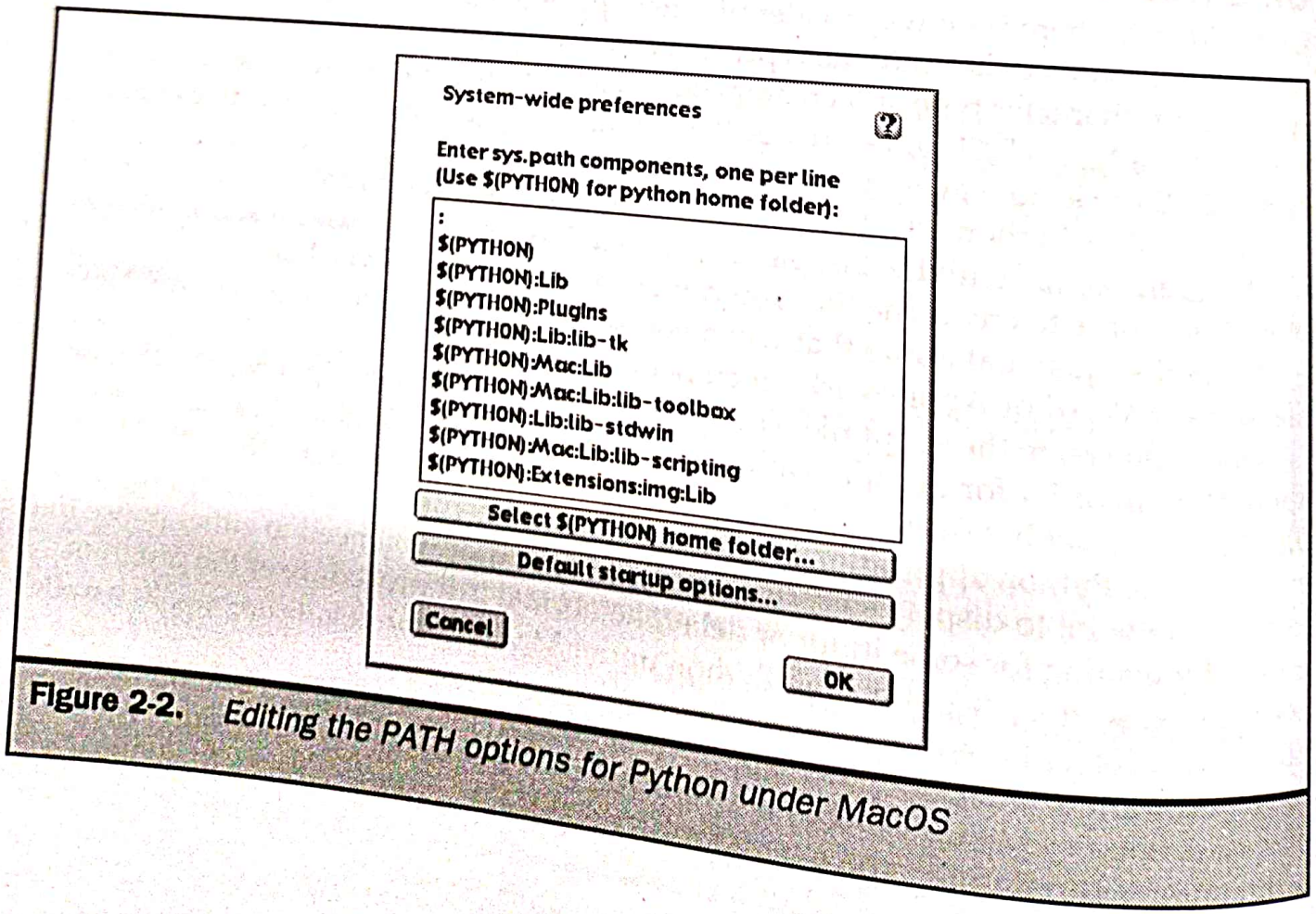



**Figure 2-2.** Editing the PATH options for Python under MacOS

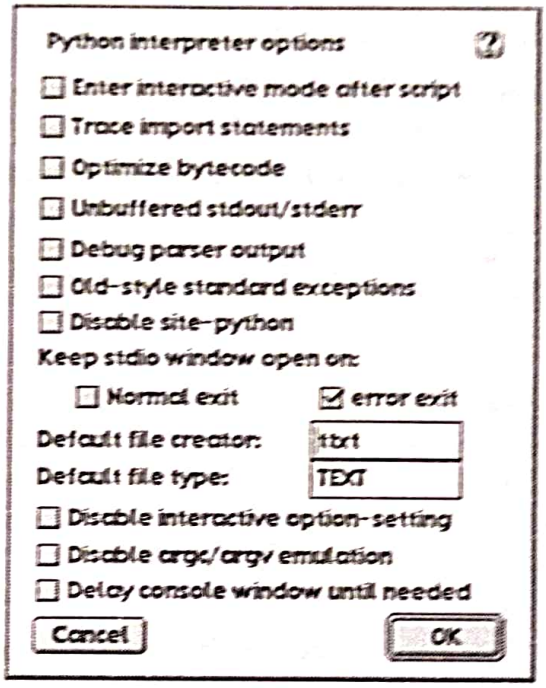



**Figure 2-3.** *Editing other MacOS options for the Python interpreter*

Table 2-3 lists the options in the window in Figure 2-3 that match the command-line versions.

Table 2-4 describes the other options under the MacOS version of the Python interpreter.

| MacOS Option | Command-Line Equivalent |
|---|---|
| Enter Interactive Mode After Script | -I |
| Trace Import Statements | -v |
| Optimize Bytecode | -O |
| Unbuffered stdout/stderr | -u |
| Debug Parser Output | -d |
| Old-Style Standard Exceptions | -X |

**Table 2-3.** *MacOS/Command-Line Equivalent Options for the Python Interpreter*

| MacOS Option | Description |
|---|---|
| Keep stdio Window Open on Normal Exit | Keeps the console window created by the Python application open after the script exits, even if the execution was successful. |
| Keep stdio Window Open on Error Exit (default) | Keeps the console window created by the Python application open after the script exits only if there was an error or an exception. |
| Default File Creator | Specifies the four-letter code used as the file creator type when files are generated by a Python script. The creator code is used to define which application opens the file when it is double-clicked. Use the code **ttxt** for TeachText/SimpleText, **MSWD** for Microsoft Word, and **R*ch** for BBEdit. |
| Default File Type | Specifies the four-letter code to be used as the file type code for files created by Python scripts. **TEXT** is a normal text file, and **bina** is a binary file. Others are application/operating system-specific. Note that if you want to use your own code, you need to register the code with Apple before supplying a public application. |
| Disable Interactive Option-Setting | If you hold down the OPTION key when starting the interpreter, you are allowed to specify some of these options on the fly. Setting this option disables this feature. |
| Disable argc/argv Emulation | Disables the ability to supply command-line options by holding down the OPTION key while starting. |
| Delay Console Window Until Needed | If the application does not require a console window or does not use one until later in the application, you can avoid displaying an empty window by setting this option. |

**Table 2-4.** *MacOS-Specific Options for the Python Interpreter*

Note that you can also specify these options on individual Python applets (but not scripts) by dragging and dropping the applet onto the **EditPythonPrefs** application.

## Other Methods

There are, of course, methods for executing Python statements. The Python language was written as much more than just a simple interpreter. You can embed the Python interpreter into a C or C++ application, and therefore execute Python statements and even entire modules within the confines of another application. This topic is too advanced for this book so we won't go into any detail, but it is another practical and very useful way of executing Python applications.

If you want to embed Python into an application, see Chapter 28 for more information. Alternatively, check the documentation that comes with Python. The standard documentation set includes the manual *Extending and Embedding the Python Interpreter*, which should contain all the information you need. See Appendix B for information on how to obtain and use the Python documentation.

## Script, Program, or Module?

New users to the Python language sometimes get confused about the difference between the different names for Python applications. Is a Python application a script, program, module, or is it something else?

Strictly speaking, a text file that contains Python statements should be referred to as a *module*. Modules within Python have special significance; they can be executed and they can also be imported by other modules. For some users this is doubly confusing because they are used to separate terms for different types of files—importable scripts in Perl are generally called modules rather than scripts. Python doesn't distinguish between the two, and there are no special tricks for creating Python modules that can be imported.

However, Python is a scripting language, so it's also perfectly legal to call a Python module a script. In addition, a Python module that performs a particular task could also be called a program or application, since both terms apply to a file that instructs a computer to do a number of specific tasks.

At the end of the day, there is really nothing special about any of the terms, but module is the generally accepted and recognized term for files that contain any Python statements.