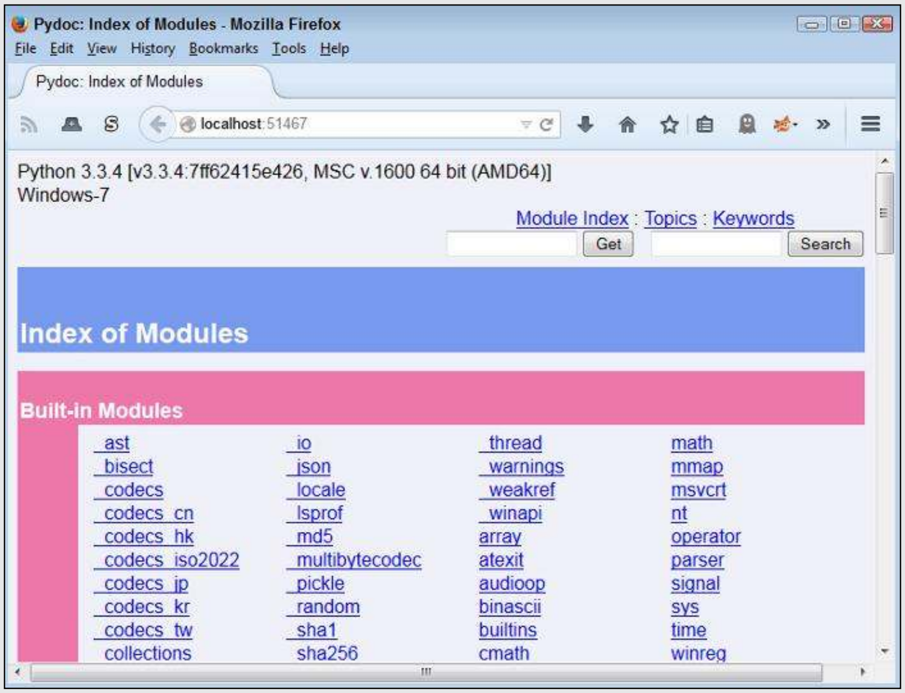
Part III

Performing Common Tasks


Pydoc: Index of Modules - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Pydoc: Index of Modules
localhost:51467

Python 3.3.4 [v3.3.4:7ff62415e426, MSC v.1600 64 bit (AMD64)]
Windows-7

Module Index : Topics : Keywords

Get Search

Index of Modules

Built-in Modules

_ast	_io	_thread	math
_bisect	_json	_warnings	mmap
_codecs	_locale	_weakref	msvcrt
_codecs_cn	_lsprof	_winapi	nt
_codecs_hk	_md5	array	operator
_codecs_iso2022	_multibytecodec	atexit	parser
_codecs_jp	_pickle	audioop	signal
_codecs_kr	_random	binascii	sys
_codecs_tw	_sha1	builtins	time
_collections	_sha256	cmath	winreg

See an example of how you can named arguments in format strings at www.dummies.com/extras/beginningprogrammingwithpython.

In this part . . .

- ✓ Gain access to Python modules.
- ✓ Slice and dice strings to meet your output needs.
- ✓ Create lists of objects you want to manage.
- ✓ Use collections to organize data efficiently.
- ✓ Develop classes to make code reusable.

Chapter 10

Interacting with Modules

In This Chapter

- Organizing your code
- Adding code from outside sources to your application
- Locating code libraries on disk
- Looking at the library code
- Obtaining and reading the Python library documentation

The examples in this book are small, but the functionality of the resulting applications is extremely limited as well. Even tiny real-world applications contain thousands of lines of code. In fact, applications that contain millions of lines of code are somewhat common. Imagine trying to work with a file large enough to contain millions of lines of code — you'd never find anything. In short, you need some method to organize code into small pieces that are easier to manage, much like the examples in this book. The Python solution is to place code in separate code groupings called *modules.* Commonly used modules that contain source code for generic needs are called *libraries.*

Modules are contained in separate files. In order to use the module, you must tell Python to grab the file and read it into the current application. The process of obtaining code found in external files is called *importing.* You import a module or library to use the code it contains. A few examples in the book have already shown the `import` statement in use, but this chapter explains the `import` statement in detail so that you know how to use it.

As part of the initial setup, Python created a pointer to the general-purpose libraries that it uses. That's why you can simply add an `import` statement with the name of the library and Python can find it. However, it pays to know how to locate the files on disk in case you ever need to update them or you want to add your own modules and libraries to the list of files that Python can use.

The library code is self-contained and well documented (at least in most cases it is). Some developers might feel that they never need to look at the library code, and they're right to some degree — you never have to look at the library code in order to use it. You might want to view the library code, though, to ensure that you understand how the code works. In addition, the library code can teach you new programming techniques that you might not otherwise discover. So, viewing the library code is optional, but it can be helpful.

The one thing you do need to know how to do is obtain and use the Python library documentation. This chapter shows you how to obtain and use the library documentation as part of the application-creation process.

Creating Code Groupings

It's important to group like pieces of code together to make the code easier to use, modify, and understand. As an application grows, managing the code found in a single file becomes harder and harder. At some point, the code becomes impossible to manage because the file has become too large for anyone to work with.

REMEMBER

The term *code* is used broadly in this particular case. Code groupings can include:

- Classes
- Functions
- Variables
- Runnable code

The collection of classes, functions, variables, and runnable code within a module is known as *attributes.* A module has attributes that you access by that attribute's name. Later sections in this chapter discuss precisely how module access works.

TECHNICAL STUFF

The runnable code can actually be written in a language other than Python. For example, it's somewhat common to find modules that are written in C/C++ instead of Python. The reason that some developers use runnable code is to make the Python application faster, less resource intensive, and better able to use a particular platform's resources. However, using runnable code comes with the downside of making your application less portable (able to run on other platforms) unless you have runnable code modules for each platform that you want to support. In addition, dual-language applications can be harder to maintain because you must have developers who can speak each of the computer languages used in the application.

The most common way to create a module is to define a separate file containing the code you want to group separately from the rest of the application. For example, you might want to create a print routine that an application uses in a number of places. The print routine isn't designed to work on its own but is part of the application as a whole. You want to separate it because the application uses it in numerous places and you could potentially use the same code in another application. The ability to reuse code ranks high on the list of reasons to create modules.

To make things easier to understand, the examples in this chapter use a common module. The module doesn't do anything too amazing, but it demonstrates the principles of working with modules. Open a Python File window and create a new file named `MyLibrary.py`. Type the code found in Listing 10-1 and save it to disk. (This module also appears with the downloadable source code as `MyLibrary.py`.)

Listing 10-1: A Simple Demonstration Module

```
def SayHello(Name):
   print("Hello ", Name)
   return

def SayGoodbye(Name):
   print("Goodbye ", Name)
   return
```

The example code contains two simple functions named `SayHello()` and `SayGoodbye()`. In both cases, you supply a `Name` to print and the function prints it onscreen along with a greeting for you. At that point, the function returns control to the caller. Obviously, you normally create more complicated functions, but these functions work well for the purposes of this chapter.

Importing Modules

In order to use a module, you must import it. Python places the module code inline with the rest of your application in memory — as if you had created one huge file. Neither file is changed on disk — they're still separate, but the way Python views the code is different.

You have two ways to import modules. Each technique is used in specific circumstances:

- `import`: You use the `import` statement when you want to import an entire module. This is the most common method that developers use to import modules because it saves time and requires only one line of code. However, this approach also uses more memory resources than does the approach of selectively importing the attributes you need, which the next paragraph describes.
- `from...import`: You use the `from...import` statement when you want to selectively import individual module attributes. This method saves resources, but at the cost of complexity. In addition, if you try to use an attribute that you didn't import, Python registers an error. Yes, the module still contains the attribute, but Python can't see it because you didn't import it.

Now that you have a better idea of how to import modules, it's time to look at them in detail. The following sections help you work through importing modules using the two techniques available in Python.

Changing the current Python directory

The directory that Python is using to access code affects which modules you can load. The Python library files are always included in the list of locations that Python can access, but Python knows nothing of the directory you use to hold your source code unless you tell it to look there. The easiest method for accomplishing this task is to change the current Python directory to point to your code folder using these steps:

1. **Open the Python Shell.**

   You see the Python Shell window appear.

2. **Type** import os **and press Enter.**

   This action imports the Python os library. You need to import this library to change the directory (the location Python sees on disk) to the working directory for this book.

3. **Type** os.chdir("C:\BP4D\Chapter10") **and press Enter.**

   You need to use the directory that contains the downloadable source or your own project files on your local hard drive. The book uses the default book directory described in Chapter 4. Python can now use the downloadable source code directory to access modules that you create for this chapter.

Using the import statement

The `import` statement is the most common method for importing a module into Python. This approach is fast and ensures that the entire module is ready for use. The following steps get you started using the `import` statement.

1. **Open the Python Shell.**

   You see the Python Shell window appear.

2. **Change directories to the downloadable source code directory.**

   See the instructions found in the "Changing the current Python directory" sidebar.

3. **Type** import MyLibrary **and press Enter.**

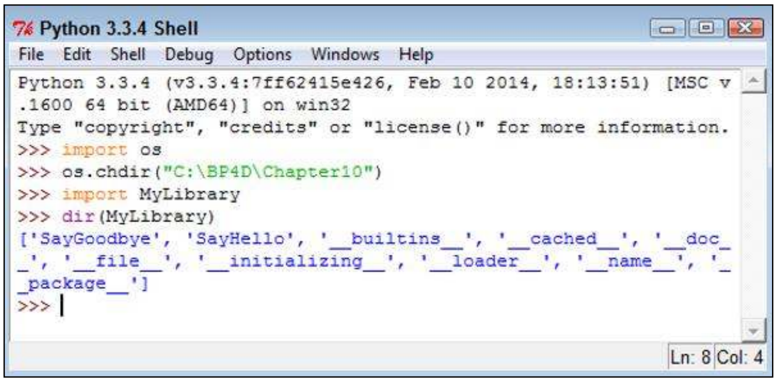
   Python imports the contents of the `MyLibrary.py` file that you created in the "Creating Code Groupings" section of the chapter. The entire library is now ready for use.

   REMEMBER

   It's important to know that Python also creates a cache of the module in the `__pycache__` subdirectory. If you look into your source code directory after you import `MyLibrary` for the first time, you see the new `__pycache__` directory. If you want to make changes to your module, you must delete this directory. Otherwise, Python will continue to use the unchanged cache file instead of your updated source code file.

4. **Type** dir(MyLibrary) **and press Enter.**

   You see a listing of the module contents, which includes the `SayHello()` and `SayGoodbye()` functions, as shown in Figure 10-1. (A discussion of the other entries appears in the "Viewing the Module Content" section of the chapter.)

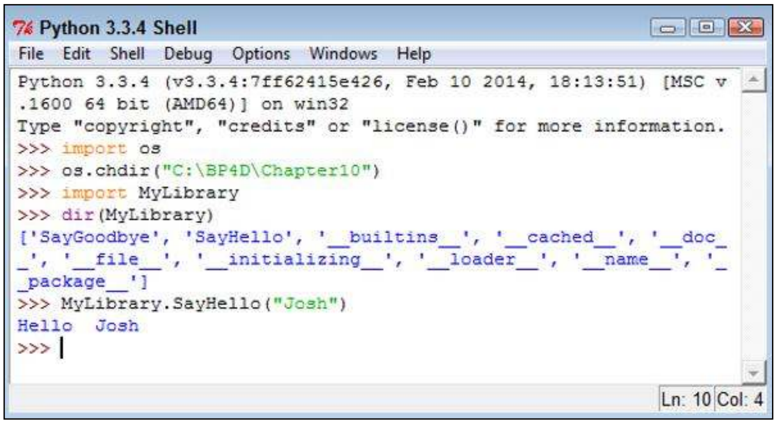


```
Python 3.3.4 (v3.3.4:7ff62415e426, Feb 10 2014, 18:13:51) [MSC v
.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import os
>>> os.chdir("C:\BP4D\Chapter10")
>>> import MyLibrary
>>> dir(MyLibrary)
['SayGoodbye', 'SayHello', '__builtins__', '__cached__', '__doc_
_', '__file__', '__initializing__', '__loader__', '__name__', '_
_package__']
>>> 
```

Figure 10-1: A directory listing shows that Python imports both functions from the module.

5. **Type** MyLibrary.SayHello("Josh") **and press Enter.**

   The `SayHello()` function outputs the expected text, as shown in Figure 10-2.



```
Python 3.3.4 (v3.3.4:7ff62415e426, Feb 10 2014, 18:13:51) [MSC v
.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import os
>>> os.chdir("C:\BP4D\Chapter10")
>>> import MyLibrary
>>> dir(MyLibrary)
['SayGoodbye', 'SayHello', '__builtins__', '__cached__', '__doc_
_', '__file__', '__initializing__', '__loader__', '__name__', '_
_package__']
>>> MyLibrary.SayHello("Josh")
Hello  Josh
>>> |
```

Figure 10-2: The `SayHello()` function outputs the expected greeting.

Notice that you must precede the attribute name, which is the `SayHello()` function in this case, with the module name, which is `MyLibrary`. The two elements are separated by a period. Every call to a module that you import follows the same pattern.

6. **Type** MyLibrary.SayGoodbye("Sally") **and press Enter.**

   The `SayGoodbye( )` function outputs the expected text.

7. **Close the Python Shell.**

   The Python Shell window closes.

Using the from...import statement

The `from...import` statement has the advantage of importing only the attributes you need from a module. This difference means that the module uses less memory and other system resources than using the `import` statement does. In addition, the `from...import` statement makes the module a little easier to use because some commands, such as `dir()`, show less information, or only the information that you actually need. The point is that you get only what you want and not anything else. The following steps demonstrate using the `from...import` statement.

1. **Open the Python Shell.**

   You see the Python Shell window appear.

2. **Change directories to the downloadable source code directory.**

   See the instructions found in the "Changing the current Python directory" sidebar.

3. **Type** from MyLibrary import SayHello **and press Enter.**

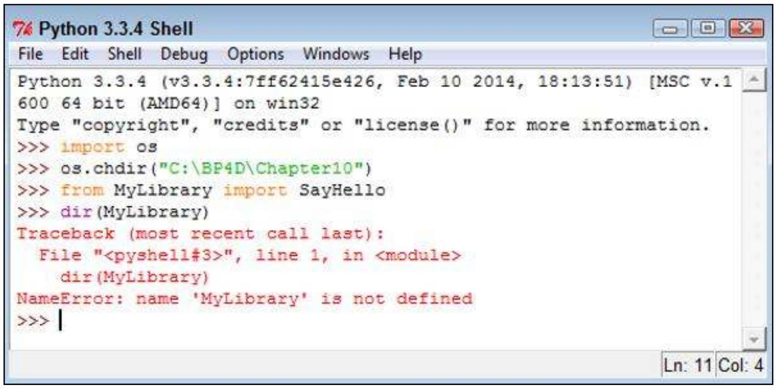
   Python imports the `SayHello()` function that you create in the "Creating Code Groupings" section, earlier in the chapter. Only this specific function is now ready for use.

   REMEMBER

   You can still import the entire module, should you want to do so. The two techniques for accomplishing the task are to create a list of modules to import (the names can be separated by commas, such as `from MyLibrary import SayHello, SayGoodbye`) or to use the asterisk (*) in place of a specific attribute name. The asterisk acts as a wildcard character that imports everything.

4. **Type** dir(MyLibrary) **and press Enter.**

   Python displays an error message, as shown in Figure 10-3. Python imports only the attributes that you specifically request. This means that the `MyLibrary` module isn't in memory — only the attributes that you requested are in memory.

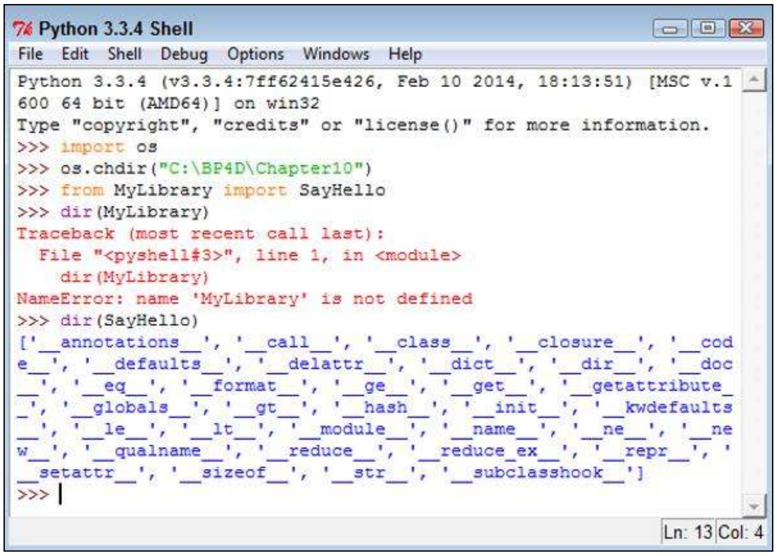


```
Python 3.3.4 (v3.3.4:7ff62415e426, Feb 10 2014, 18:13:51) [MSC v.1
600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import os
>>> os.chdir("C:\BP4D\Chapter10")
>>> from MyLibrary import SayHello
>>> dir(MyLibrary)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    dir(MyLibrary)
NameError: name 'MyLibrary' is not defined
>>> 
```

Figure 10-3: The `from...import` statement imports only the items that you specifically request.

5. **Type** dir(SayHello) **and press Enter.**

   You see a listing of attributes that are associated with the `SayHello()` function, as shown in Figure 10-4. It isn't important to know how these attributes work just now, but you'll use some of them later in the book.

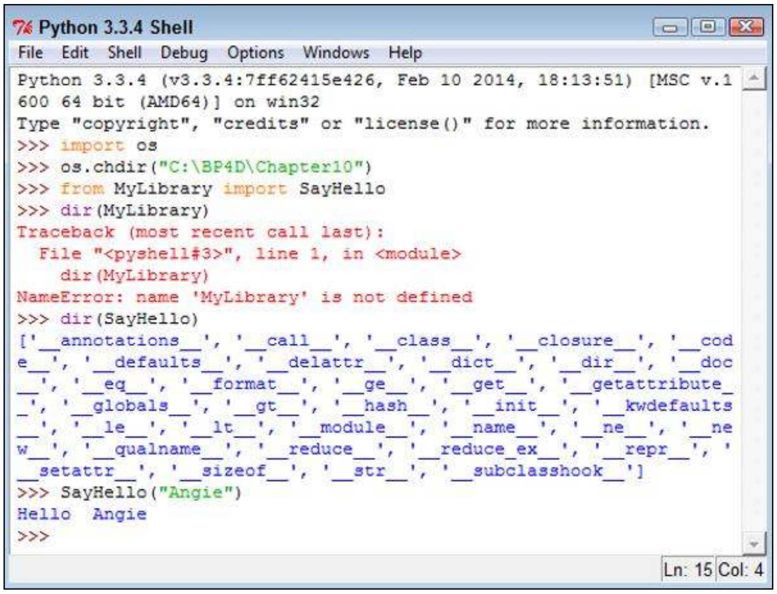


```
Python 3.3.4 (v3.3.4:7ff62415e426, Feb 10 2014, 18:13:51) [MSC v.1
600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import os
>>> os.chdir("C:\BP4D\Chapter10")
>>> from MyLibrary import SayHello
>>> dir(MyLibrary)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    dir(MyLibrary)
NameError: name 'MyLibrary' is not defined
>>> dir(SayHello)
['__annotations__', '__call__', '__class__', '__closure__', '__cod
e__', '__defaults__', '__delattr__', '__dict__', '__dir__', '__doc
__', '__eq__', '__format__', '__ge__', '__get__', '__getattribute_
_', '__globals__', '__gt__', '__hash__', '__init__', '__kwdefaults
__', '__le__', '__lt__', '__module__', '__name__', '__ne__', '__ne
w__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '
__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>> |
```

Figure 10-4: Use the `dir()` function to obtain information about the specific attributes you import.

6. **Type** SayHello("Angie") **and press Enter.**

   The `SayHello()` function outputs the expected text, as shown in Figure 10-5.



```
Python 3.3.4 (v3.3.4:7ff62415e426, Feb 10 2014, 18:13:51) [MSC v.1
600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import os
>>> os.chdir("C:\BP4D\Chapter10")
>>> from MyLibrary import SayHello
>>> dir(MyLibrary)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    dir(MyLibrary)
NameError: name 'MyLibrary' is not defined
>>> dir(SayHello)
['__annotations__', '__call__', '__class__', '__closure__', '__cod
e__', '__defaults__', '__delattr__', '__dict__', '__dir__', '__doc
__', '__eq__', '__format__', '__ge__', '__get__', '__getattribute_
_', '__globals__', '__gt__', '__hash__', '__init__', '__kwdefaults
__', '__le__', '__lt__', '__module__', '__name__', '__ne__', '__ne
w__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '
__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>> SayHello("Angie")
Hello  Angie
>>>
```

Figure 10-5: The `SayHello()` function no longer requires the module name.

When you import attributes using the `from...import` statement, you don't need to precede the attribute name with a module name. This feature makes the attribute easier to access.

Using the `from...import` statement can also cause problems. If two attributes have the same name, you can import only one of them. The `import` statement prevents name collisions, which is important when you have a large number of attributes to import. In sum, you must exercise care when using the `from...import` statement.

7. **Type** SayGoodbye("Harold") **and press Enter.**

   You imported only the `SayHello()` function, so Python knows nothing about `SayGoodbye()` and displays an error message. The selective nature of the `from...import` statement can cause problems when you assume that an attribute is present when it really isn't.

8. **Close the Python Shell.**

   The Python Shell window closes.

Finding Modules on Disk

In order to use the code in a module, Python must be able to locate the module and load it into memory. The location information is stored as paths within Python. Whenever you request that Python import a module, Python looks at all the files in its list of paths to find it. The path information comes from three sources:

- **Environment variables:** Chapter 3 tells you about Python environment variables, such as `PYTHONPATH`, that tell Python where to find modules on disk.
- **Current directory:** Earlier in this chapter, you discover that you can change the current Python directory so that it can locate any modules used by your application.
- **Default directories:** Even when you don't define any environment variables and the current directory doesn't yield any usable modules, Python can still find its own libraries in the set of default directories that are included as part of its own path information.

It's helpful to know the current path information because the lack of a path can cause your application to fail. The following steps demonstrate how you can obtain path information:

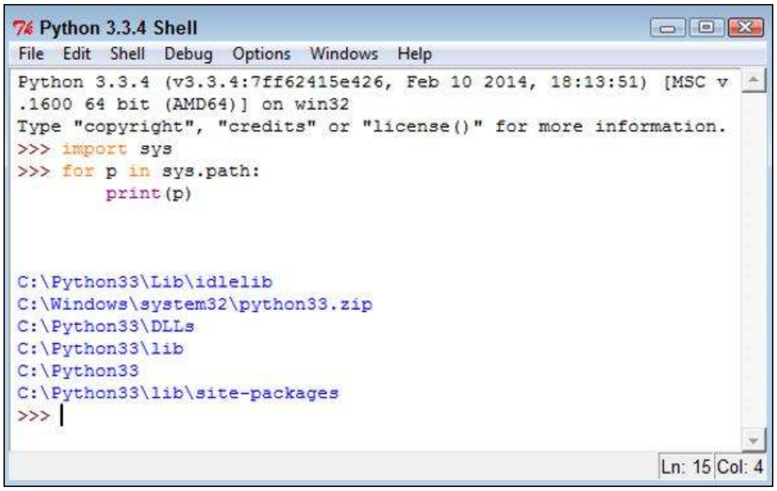
1. **Open the Python Shell.**

   You see the Python Shell window appear.

2. **Type** import sys **and press Enter.**
3. **Type** for p in sys.path: **and press Enter.**

   Python automatically indents the next line for you. The `sys.path` attribute always contains a listing of default paths.
4. **Type** print(p) **and press Enter twice.**

   You see a listing of the path information, as shown in Figure 10-6. Your listing may be different from the one shown in Figure 10-6, depending on your platform, the version of Python you have installed, and the Python features you have installed.



```
Python 3.3.4 (v3.3.4:7ff62415e426, Feb 10 2014, 18:13:51) [MSC v
.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import sys
>>> for p in sys.path:
        print(p)



C:\Python33\Lib\idlelib
C:\Windows\system32\python33.zip
C:\Python33\DLLs
C:\Python33\lib
C:\Python33
C:\Python33\lib\site-packages
>>> 
```

Figure 10-6: The `sys.path` attribute contains a listing of the individual paths for your system.

The `sys.path` attribute is reliable but may not always contain every path that Python can see. If you don't see a needed path, you can always check in another place that Python looks for information. The following steps show how to perform this task:

1. **Type** import os **and press Enter.**
2. **Type** os.environ['PYTHONPATH'].split(os.pathsep) **and press Enter.**

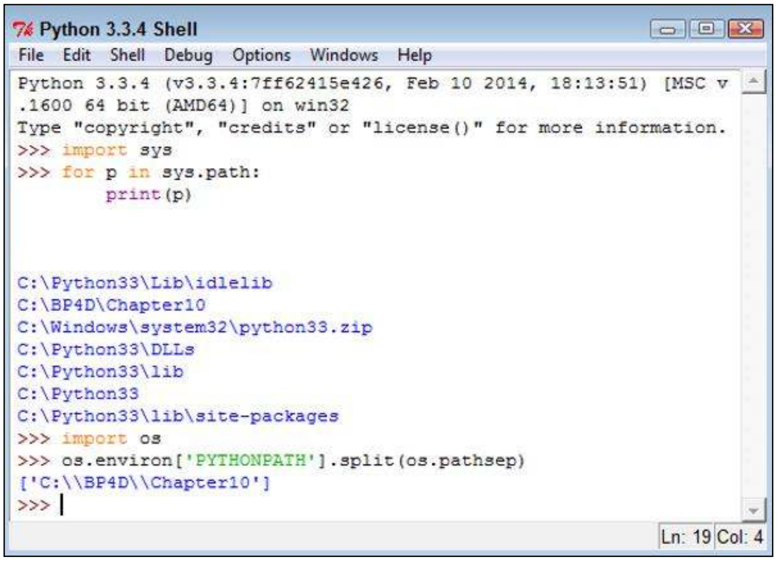
   When you have a `PYTHONPATH` environment variable defined, you see a list of paths, as shown in Figure 10-7. However, if you don't have the environment variable defined, you see an error message instead.

   Notice that both the `sys.path` and the `os.environ['PYTHONPATH']` attributes contain the `C:\BP4D\Chapter10` entry in this case. The `sys.path` attribute doesn't include the `split()` function, which is why the example uses a `for` loop with it. However, the `os.environ['PYTHONPATH']` attribute does include the `split()` function, so you can use it to create a list of individual paths.

You must provide `split()` with a value to look for in splitting a list of items. The `os.pathsep` *constant* (a variable that has one, unchangeable, defined value) defines the path separator for the current platform so that you can use the same code on any platform that supports Python.

3. **Close the Python Shell.**

   The Python Shell window closes.



```
Python 3.3.4 (v3.3.4:7ff62415e426, Feb 10 2014, 18:13:51) [MSC v
.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import sys
>>> for p in sys.path:
        print(p)



C:\Python33\Lib\idlelib
C:\BP4D\Chapter10
C:\Windows\system32\python33.zip
C:\Python33\DLLs
C:\Python33\lib
C:\Python33
C:\Python33\lib\site-packages
>>> import os
>>> os.environ['PYTHONPATH'].split(os.pathsep)
['C:\\BP4D\\Chapter10']
>>> 
```

Figure 10-7: You must request information about environment variables separately.

TIP

You can also add and remove items from `sys.path`. For example, if you want to add Chapter 9 to the list of modules, you type **`sys.path.append("C:\\BP4D\\Chapter09")`** and press Enter in the Python Shell window. When you list the `sys.path` contents again, you see that the new entry is added. Likewise, when you want to remove an entry, such as Chapter 9, you type **`sys.path.remove("C:\\BP4D\\Chapter09")`** and press Enter.

Viewing the Module Content

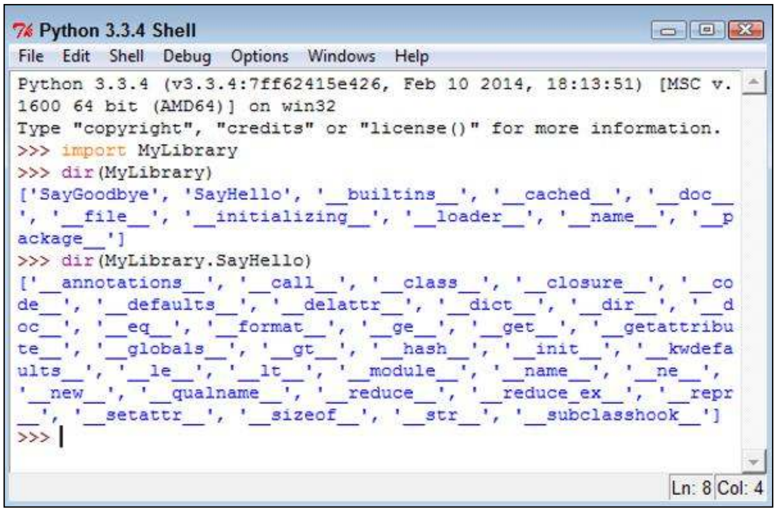
Python gives you several different ways to view module content. The method that most developers use is to work with the `dir()` function, which tells you about the attributes that the module provides.

Look at Figure 10-1, earlier in the chapter. In addition to the `SayGoodbye()` and `SayHello()` function entries discussed previously, the list has other entries. These attributes are automatically generated by Python for you. These attributes perform the following tasks or contain the following information:

- `__builtins__`: Contains a listing of all the built-in attributes that are accessible from the module. Python adds these attributes automatically for you.
- `__cached__`: Tells you the name and location of the cached file that is associated with the module. The location information (path) is relative to the current Python directory.
- `__doc__`: Outputs help information for the module, assuming that you've actually filled it in. For example, if you type **`os.__doc__`** and press Enter, Python will output the help information associated with the `os` library.
- `__file__`: Tells you the name and location of the module. The location information (path) is relative to the current Python directory.
- `__initializing__`: Determines whether the module is in the process of initializing itself. Normally this attribute returns a value of `False`. This attribute is useful when you need to wait until one module is done loading before you import another module that depends on it.
- `__loader__`: Outputs the loader information for this module. The *loader* is a piece of software that gets the module and puts it into memory so that Python can use it. This is one attribute you rarely (if ever) use.
- `__name__`: Tells you just the name of the module.
- `__package__`: This attribute is used internally by the import system to make it easier to load and manage modules. You don't need to worry about this particular attribute.

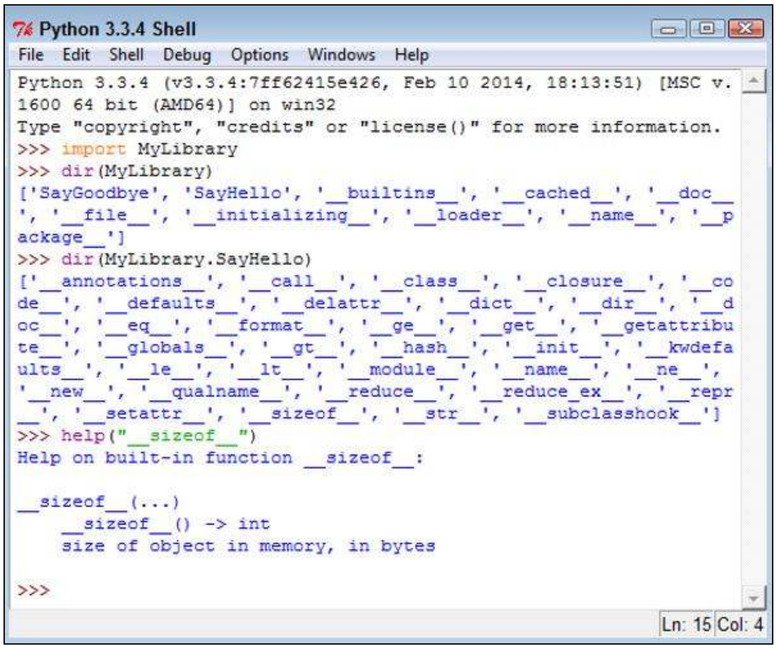
It may surprise you to find that you can drill down even further into the attributes. Type **dir(MyLibrary.SayHello)** and press Enter. You see the entries shown in Figure 10-8.

Some of these entries, such as `__name__`, also appeared in the module listing. However, you might be curious about some of the other entries. For example, you might want to know what `__sizeof__` is all about. One way to get additional information is to type **help("__sizeof__")** and press Enter. You see some scanty (but useful) help information, as shown in Figure 10-9.



```
Python 3.3.4 (v3.3.4:7ff62415e426, Feb 10 2014, 18:13:51) [MSC v.
1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import MyLibrary
>>> dir(MyLibrary)
['SayGoodbye', 'SayHello', '__builtins__', '__cached__', '__doc__
', '__file__', '__initializing__', '__loader__', '__name__', '__p
ackage__']
>>> dir(MyLibrary.SayHello)
['__annotations__', '__call__', '__class__', '__closure__', '__co
de__', '__defaults__', '__delattr__', '__dict__', '__dir__', '__d
oc__', '__eq__', '__format__', '__ge__', '__get__', '__getattribu
te__', '__globals__', '__gt__', '__hash__', '__init__', '__kwdefa
ults__', '__le__', '__lt__', '__module__', '__name__', '__ne__',
'__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr
__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>> 
```

Figure 10-8: Drill down as far as needed to understand the modules that you use in Python.



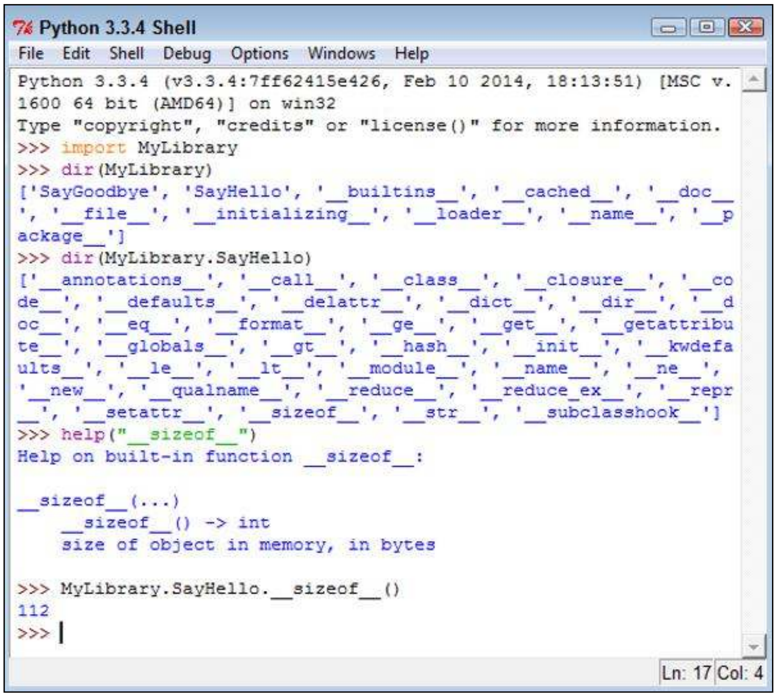
```
Python 3.3.4 (v3.3.4:7ff62415e426, Feb 10 2014, 18:13:51) [MSC v.
1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import MyLibrary
>>> dir(MyLibrary)
['SayGoodbye', 'SayHello', '__builtins__', '__cached__', '__doc__
', '__file__', '__initializing__', '__loader__', '__name__', '__p
ackage__']
>>> dir(MyLibrary.SayHello)
['__annotations__', '__call__', '__class__', '__closure__', '__co
de__', '__defaults__', '__delattr__', '__dict__', '__dir__', '__d
oc__', '__eq__', '__format__', '__ge__', '__get__', '__getattribu
te__', '__globals__', '__gt__', '__hash__', '__init__', '__kwdefa
ults__', '__le__', '__lt__', '__module__', '__name__', '__ne__',
'__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr
__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>> help("__sizeof__")
Help on built-in function __sizeof__:

__sizeof__(...)
    __sizeof__() -> int
    size of object in memory, in bytes

>>> 
```

Figure 10-9: Try getting some help information about the attribute you want to know about.

Python isn't going to blow up if you try the attribute. Even if the shell does experience problems, you can always start a new one. So, another way to check out a module is to simply try the attributes. For example, if you type **MyLibrary.SayHello.__sizeof__()** and press Enter, you see the size of the `SayHello()` function in bytes, as shown in Figure 10-10.



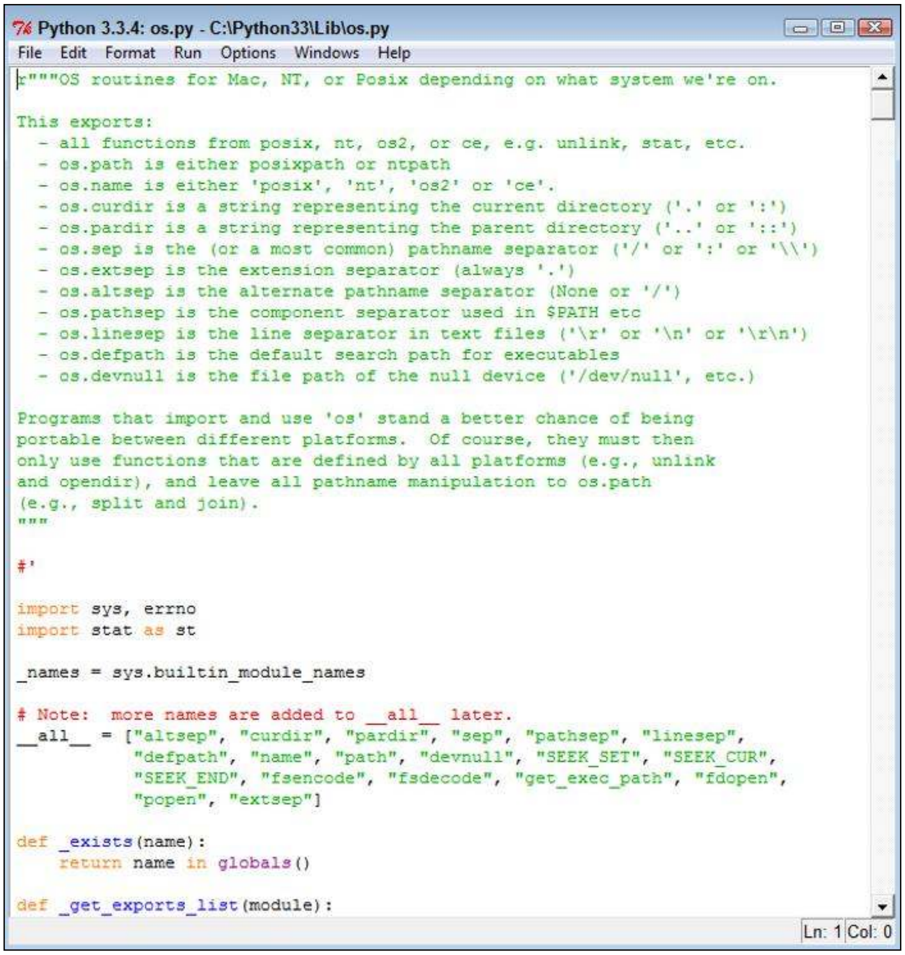
```
Python 3.3.4 (v3.3.4:7ff62415e426, Feb 10 2014, 18:13:51) [MSC v.
1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import MyLibrary
>>> dir(MyLibrary)
['SayGoodbye', 'SayHello', '__builtins__', '__cached__', '__doc__
', '__file__', '__initializing__', '__loader__', '__name__', '__p
ackage__']
>>> dir(MyLibrary.SayHello)
['__annotations__', '__call__', '__class__', '__closure__', '__co
de__', '__defaults__', '__delattr__', '__dict__', '__dir__', '__d
oc__', '__eq__', '__format__', '__ge__', '__get__', '__getattribu
te__', '__globals__', '__gt__', '__hash__', '__init__', '__kwdefa
ults__', '__le__', '__lt__', '__module__', '__name__', '__ne__',
'__new__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr
__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>> help("__sizeof__")
Help on built-in function __sizeof__:

__sizeof__(...)
    __sizeof__() -> int
    size of object in memory, in bytes

>>> MyLibrary.SayHello.__sizeof__()
112
>>> |
```

Figure 10-10: Using the attributes will help you get a better feel for how they work.

Unlike many other programming languages, Python also makes the source code for its native language libraries available. For example, when you look into the `\Python33\Lib` directory, you see a listing of `.py` files that you can open in IDLE with no problem at all. Try opening the `os.py` library that you use for various tasks in this chapter, and you see the content shown in Figure 10-11.


Python 3.3.4: os.py - C:\Python33\Lib\os.py
File Edit Format Run Options Windows Help
Ln: 1 Col: 0

```
r"""OS routines for Mac, NT, or Posix depending on what system we're on.

This exports:
  - all functions from posix, nt, os2, or ce, e.g. unlink, stat, etc.
  - os.path is either posixpath or ntpath
  - os.name is either 'posix', 'nt', 'os2' or 'ce'.
  - os.curdir is a string representing the current directory ('.' or ':')
  - os.pardir is a string representing the parent directory ('..' or '::')
  - os.sep is the (or a most common) pathname separator ('/' or ':' or '\\')
  - os.extsep is the extension separator (always '.')
  - os.altsep is the alternate pathname separator (None or '/')
  - os.pathsep is the component separator used in $PATH etc
  - os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
  - os.defpath is the default search path for executables
  - os.devnull is the file path of the null device ('/dev/null', etc.)

Programs that import and use 'os' stand a better chance of being
portable between different platforms.  Of course, they must then
only use functions that are defined by all platforms (e.g., unlink
and opendir), and leave all pathname manipulation to os.path
(e.g., split and join).
"""

#'

import sys, errno
import stat as st

_names = sys.builtin_module_names

# Note:  more names are added to __all__ later.
__all__ = ["altsep", "curdir", "pardir", "sep", "pathsep", "linesep",
           "defpath", "name", "path", "devnull", "SEEK_SET", "SEEK_CUR",
           "SEEK_END", "fsencode", "fsdecode", "get_exec_path", "fdopen",
           "popen", "extsep"]

def _exists(name):
    return name in globals()

def _get_exports_list(module):
```

Figure 10-11: Directly viewing module code can help in understanding it.

Viewing the content directly can help you discover new programming techniques and better understand how the library works. The more time you spend working with Python, the better you'll become at using it to build interesting applications.

WARNING!

Make sure that you just look at the library code and don't accidentally change it. If you accidentally change the code, your applications can stop working. Worse yet, you can introduce subtle bugs into your application that will appear only on your system and nowhere else. Always exercise care when working with library code.

Using the Python Module Documentation

You can use the `doc()` function whenever needed to get quick help. However, you have a better way to study the modules and libraries located in the Python path — the Python Module Documentation. This feature often appears as Module Docs in the Python folder on your system. It's also referred to as pydoc. Whatever you call it, the Python Module Documentation makes life a lot easier for developers. The following sections describe how to work with this feature.

Opening the pydoc application

Pydoc is just another Python application. It actually appears in the `\Python33\Lib` directory of your system as `pydoc.py`. As with any other `.py` file, you can open this one with IDLE and study how it works. You can start it using the Module Docs shortcut that appears in the Python folder on your system or by using a command at the command prompt.

The application creates a localized server that works with your browser to display information about the Python modules and libraries. So, when you start this application, you see a command (terminal) window open like the one shown in Figure 10-12.

Accessing pydoc on Windows

The Windows installation of Python has a problem. When you click Module Docs, nothing happens. Of course, this is a bit disconcerting because users are apt to feel that something is wrong with their systems or with Python itself. It turns out that the shortcut is faulty. To overcome this problem, you must create a new shortcut using the following steps:

1. **Right-click the Desktop and choose New➪Shortcut from the context menu.**

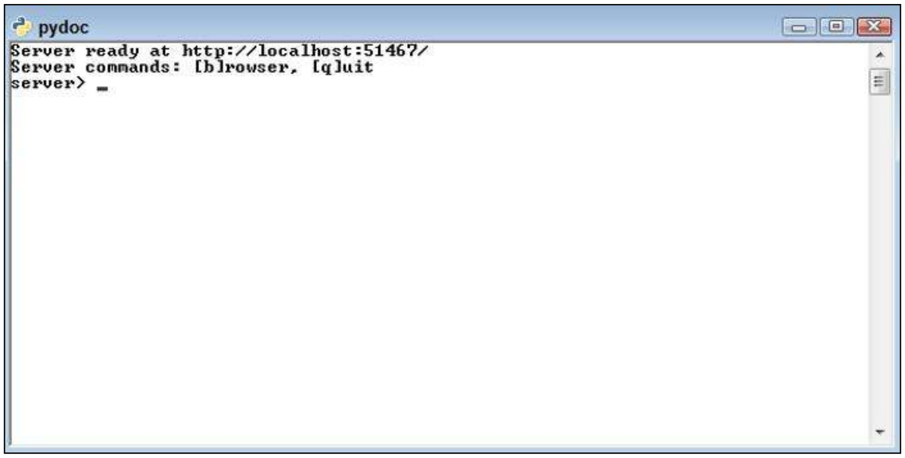
   You see the Create Shortcut wizard.

2. **Type** C:\Python33\python.exe C:\Python33\Lib\pydoc.py -b **and click Next.**

   This command line starts a copy of the pydoc server so that you can access module information.

3. **Type** pydoc **and click Finish.**

   Windows creates a new shortcut for you. This shortcut allows you to access the module help information that currently doesn't work with Python 3.3.4 on Windows.



```
Server ready at http://localhost:51467/
Server commands: [b]rowser, [q]uit
server> _
```

Figure 10-12: Starting pydoc means opening a command (terminal) window to start the server.

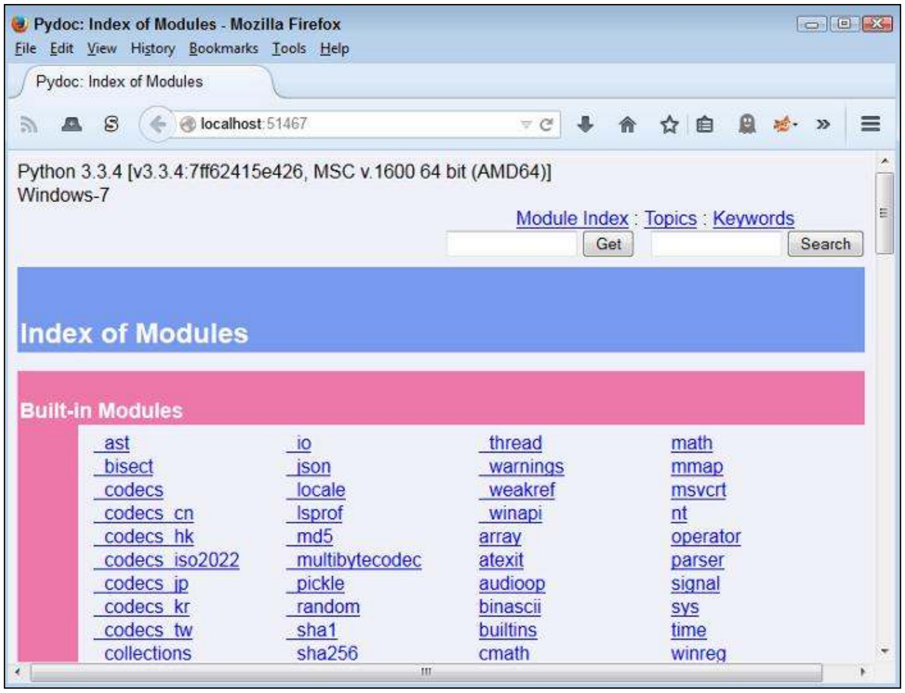
As with any server, your system may prompt you for permissions. For example, you may see a warning from your firewall telling you that pydoc is attempting to access the local system. You need to give pydoc permission to work with the system so that you can see the information it provides. Any virus detection that you have installed may need permission to let pydoc continue as well. Some platforms, such as Windows, may require an elevation in privileges to run pydoc.

Normally, the server automatically opens a new browser window for you, as shown in Figure 10-13. This window contains links to the various modules that are contained on your system, including any custom modules you create and include in the Python path. To see information about any module, you can simply click its link.

The command prompt provides you with two commands to control the server. You simply type the letter associated with the command and press Enter to activate it. Here are the two commands:

- `b`: Starts a new copy of the default browser with the index page loaded.
- `q`: Stops the server.

When you're done browsing the help information, make sure that you stop the server by typing `q` and pressing Enter at the command prompt. Stopping the server frees any resources it uses and closes any holes you made in your firewall to accommodate pydoc.


Pydoc: Index of Modules - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Pydoc: Index of Modules
localhost:51467

Python 3.3.4 [v3.3.4:7ff62415e426, MSC v.1600 64 bit (AMD64)]
Windows-7

Module Index : Topics : Keywords

Get Search

Index of Modules

Built-in Modules

_ast	_io	_thread	math
_bisect	_json	_warnings	mmap
_codecs	_locale	_weakref	msvcrt
_codecs_cn	_lsprof	_winapi	nt
_codecs_hk	_md5	array	operator
_codecs_iso2022	_multibytecodec	atexit	parser
_codecs_jp	_pickle	audioop	signal
_codecs_kr	_random	binascii	sys
_codecs_tw	_sha1	builtins	time
_collections	_sha256	cmath	winreg

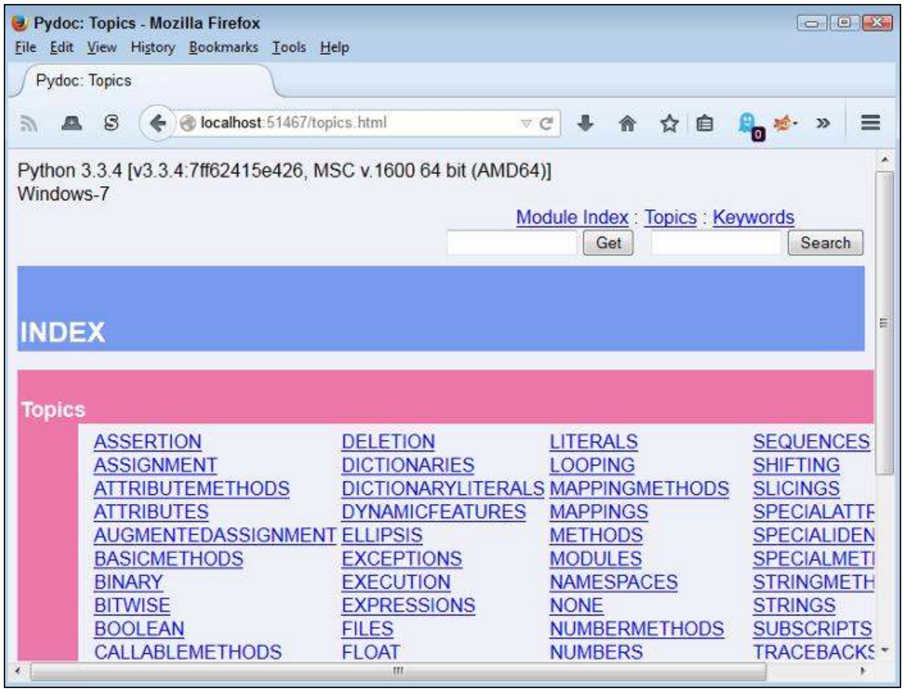
Figure 10-13: Your browser displays a number of links that appear as part of the Index page.

Using the quick-access links

Refer back to Figure 10-13. Near the top of the page, you see three links. These links provide quick access to the site features. The browser always begins at the Module Index. If you need to return to this page, simply click the Module Index link.

The Topics link takes you to the page shown in Figure 10-14. This page contains links for essential Python topics. For example, if you want to know more about Boolean values, click the BOOLEAN link. The page you see next describes how Boolean values work in Python. At the bottom of the page are related links that lead to pages that contain additional helpful information.

The Keywords link takes you to the page shown in Figure 10-15. What you see is a list of the keywords that Python supports. For example, if you want to know more about creating `for` loops, you click the for link.

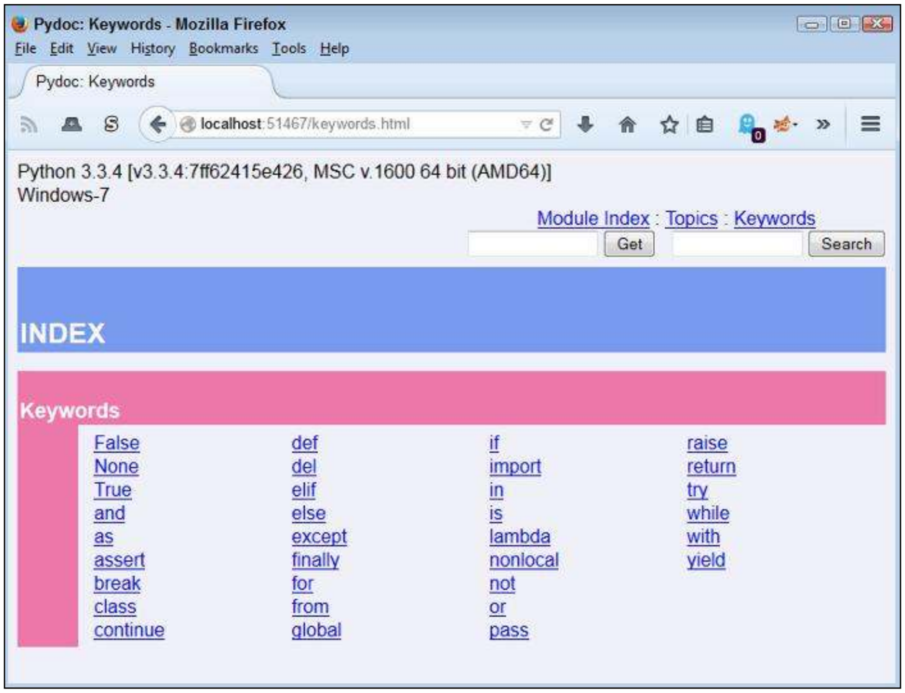


Pydoc: Topics - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Pydoc: Topics
localhost:51467/topics.html
Module Index : Topics : Keywords
Get
Search
INDEX
Topics

Python 3.3.4 [v3.3.4:7ff62415e426, MSC v.1600 64 bit (AMD64)]
Windows-7

ASSERTION	DELETION	LITERALS	SEQUENCES
ASSIGNMENT	DICTIONARIES	LOOPING	SHIFTING
ATTRIBUTEMETHODS	DICTIONARYLITERALS	MAPPINGMETHODS	SLICINGS
ATTRIBUTES	DYNAMICFEATURES	MAPPINGS	SPECIALATTF
AUGMENTEDASSIGNMENT	ELLIPSIS	METHODS	SPECIALIDEN
BASICMETHODS	EXCEPTIONS	MODULES	SPECIALMETI
BINARY	EXECUTION	NAMESPACES	STRINGMETH
BITWISE	EXPRESSIONS	NONE	STRINGS
BOOLEAN	FILES	NUMBERMETHODS	SUBSCRIPTS
CALLABLEMETHODS	FLOAT	NUMBERS	TRACEBACKS

Figure 10-14: The Topics page tells you about essential Python topics, such as how Boolean values work.



Pydoc: Keywords - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Pydoc: Keywords
localhost:51467/keywords.html
Module Index : Topics : Keywords
Get
Search
INDEX
Keywords

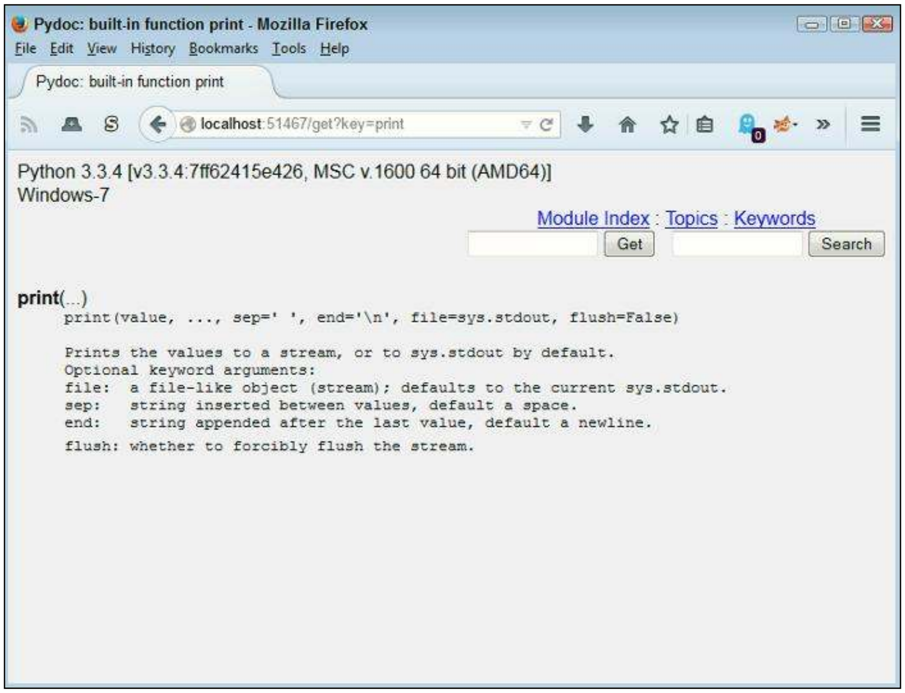
Python 3.3.4 [v3.3.4:7ff62415e426, MSC v.1600 64 bit (AMD64)]
Windows-7

False	def	if	raise
None	del	import	return
True	elif	in	try
and	else	is	while
as	except	lambda	with
assert	finally	nonlocal	yield
break	for	not	
class	from	or	
continue	global	pass	

Figure 10-15: The Keywords page contains a listing of keywords that Python supports.

Typing a search term

The pages also include two text boxes near the top. The first has a Get button next to it and the second has a Search button next to it. When you type a search term in the first text box and click Get, you see the documentation for that particular module or attribute. Figure 10-16 shows what you see when you type **print** and click Get.


Pydoc: built-in function print - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Pydoc: built-in function print
localhost:51467/get?key=print
Module Index : Topics : Keywords
Get
Search

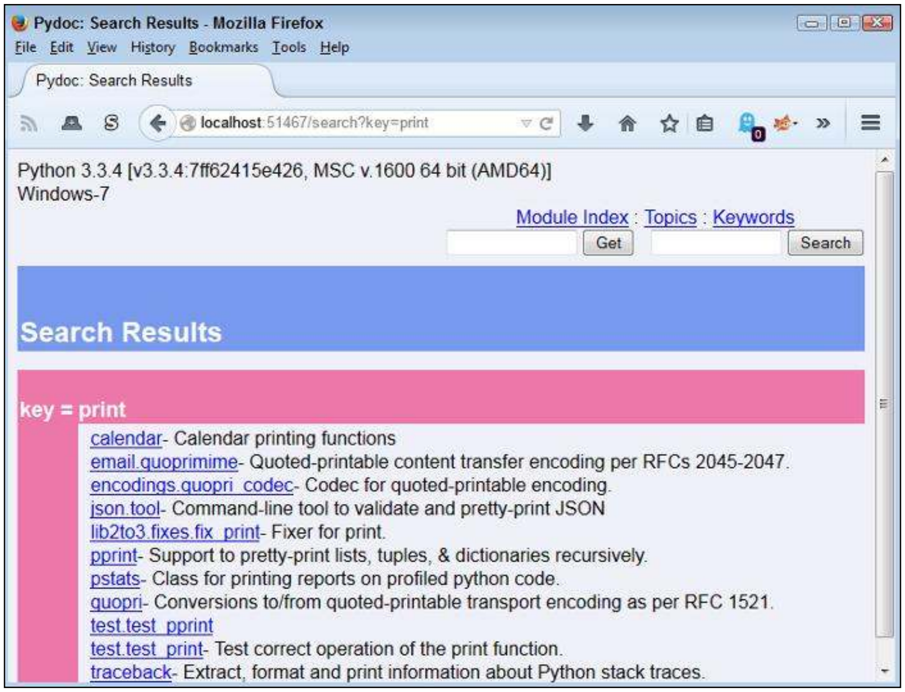
```
Python 3.3.4 [v3.3.4:7ff62415e426, MSC v.1600 64 bit (AMD64)]
Windows-7

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Figure 10-16: Using Get obtains specific information about a search term.

When you type a search term in the second text box and click Search, you see all the topics that could relate to that search term. Figure 10-17 shows typical results when you type **print** and click Search. In this case, you click a link, such as calendar, to see additional information.


Pydoc: Search Results - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Pydoc: Search Results
localhost:51467/search?key=print

Python 3.3.4 [v3.3.4:7ff62415e426, MSC v.1600 64 bit (AMD64)]
Windows-7

Module Index : Topics : Keywords
Get
Search

Search Results

key = print

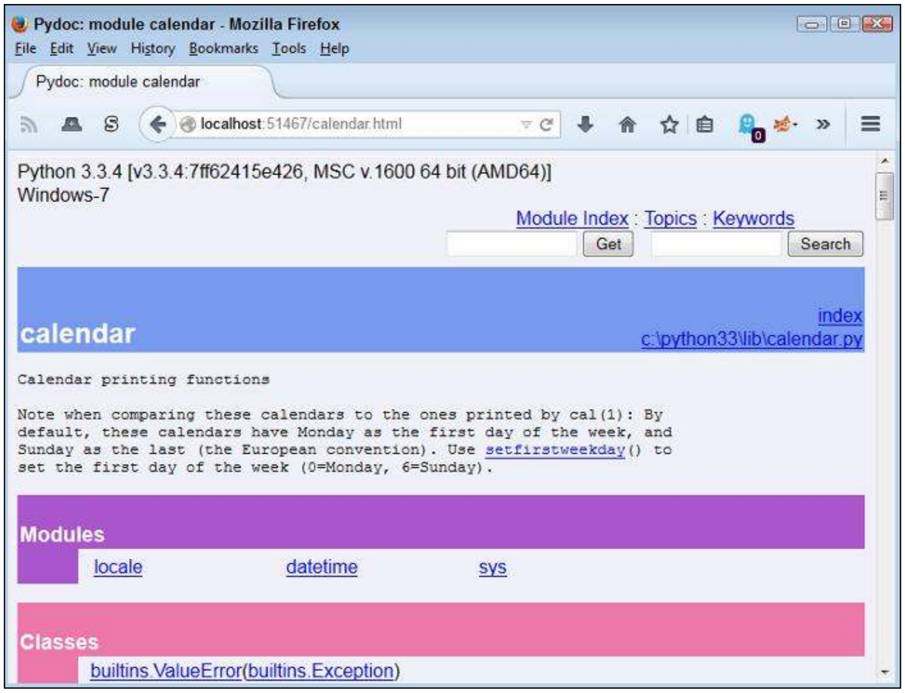
calendar- Calendar printing functions
email.quoprimime- Quoted-printable content transfer encoding per RFCs 2045-2047.
encodings.quopri_codec- Codec for quoted-printable encoding.
json.tool- Command-line tool to validate and pretty-print JSON
lib2to3.fixes.fix_print- Fixer for print.
pprint- Support to pretty-print lists, tuples, & dictionaries recursively.
pstats- Class for printing reports on profiled python code.
quopri- Conversions to/from quoted-printable transport encoding as per RFC 1521.
test.test_pprint
test.test_print- Test correct operation of the print function.
traceback- Extract, format and print information about Python stack traces.

Figure 10-17: Using Search obtains a list of topics about a search term.

Viewing the results

The results you get when you view a page depends on the topic. Some topics are brief, such as the one shown in Figure 10-16 for print. However, other topics are extensive. For example, if you were to click the calendar link in Figure 10-17, you would see a significant amount of information, as shown in Figure 10-18.

In this particular case, you see related module information, error information, functions, data, and all sorts of additional information about the calendar printing functions. The amount of information you see depends partly on the complexity of the topic and partly on the amount of information the developer provided with the module. For example, if you were to select MyLibrary from the Module Index page, you would see only a list of functions and no documentation at all.


Pydoc: module calendar - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Pydoc: module calendar
localhost:51467/calendar.html

Python 3.3.4 [v3.3.4:7ff62415e426, MSC v.1600 64 bit (AMD64)]
Windows-7

Module Index : Topics : Keywords

Get Search

calendar

index
c:\python33\lib\calendar.py

```
Calendar printing functions

Note when comparing these calendars to the ones printed by cal(1): By
default, these calendars have Monday as the first day of the week, and
Sunday as the last (the European convention). Use setfirstweekday() to
set the first day of the week (0=Monday, 6=Sunday).
```

Modules

locale datetime sys

Classes

builtins.ValueError(builtins.Exception)

Figure 10-18: Some pages contain extensive information.