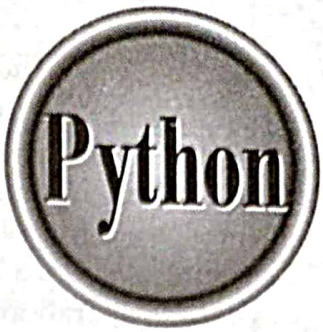
The Complete Reference



Chapter 11

Working with Files

You can't always rely on using information entirely within your application. Sometimes you need to read information from a file or write the information to a file. In other situations you need to be able to open and update an existing file in place. Editing the contents of a file is often only part of the problem. Sometimes you don't know the name of the file and need to offer the user a list of filenames. Other times you want to process a list of files matching a particular name or extension. On the periphery of these operations are requirements to make, delete, obtain, and modify files and directories, as well as their ownership and accessibility settings.

This chapter looks at all of these operations.

File Processing

The basic techniques for opening a file were described in Chapter 3 and Chapter 8. You open a file using the built-in **open()** function, which accepts at least one argument, the name of the file to open. The basic format of the **open()** function is

```
file = open(filename [, mode [, bufsize]])
```

By default, the file is opened in read mode, but an optional second argument can be used to specify the mode used to open the file, from basic reading to reading with update and binary modes. Table 11-1 lists the supported modes for opening a file.

Mode	Meaning
r	Open for reading.
w	Open for writing.
a	Open for appending (file position automatically seeks to the end during the open).
r+	Open for updating (reading and writing).
w+	Truncates (empties) the file and then opens it for reading and writing.
a+	Opens the file for reading and writing and automatically changes the current file position to the end of the file.
b	When appended to any option, opens the file in binary rather than text mode. This mode is available for Windows, DOS, and some other operating systems only. Unix/MacOS/BeOS treat all files as binary, regardless of this option.

Table 11-1 File Modes for the **open** Function

Bufsize Value	Description
0	Disable buffering.
1	Line buffered.
>1	Use a buffer that is approximately **bufsize** characters in length.
<0	Use the system default (line buffered for **tty** devices and fully buffered for any other file).

Table 11-2. *Buffer Sizes Supported by the* ***open()*** *Function*

A third optional argument defines whether the file should use buffering and if so, what size the buffer should be. Table 11-2 lists the available settings, with the default being <0.

The function returns a new file object, and it's through this file object that you read and write information to the file. Table 11-3 lists the supported methods to the file object, but you'll learn about each method in more detail later in this chapter.

Method	Description
f.close	Closes the file.
f.fileno	Returns the integer file descriptor number.
f.flush	Flushes the output buffers.
f.isatty	Returns 1 if file is an interactive terminal.
f.read([count])	Reads **count** bytes from the file.
f.readline	Reads a single line, returning it as a string.
f.readlines	Reads all the lines, returning a list of strings.
f.seek(offset [, where])	Changes the file pointer to **offset** relative to **where**.
f.tell	Gets the current file pointer.
f.truncate([size])	Truncates file to **size**.
f.write(string)	Writes **string** to the file.
f.writelines(list)	Writes the lines in **list** (a list of strings) to the file. The information in **list** is written raw; each element is not processed in any way, and the **writelines()** method does not imply that linefeed or carriage return characters are added to the output.

Table 11-3. *Methods for a File Object*

Reading

Reading information from a file is handled in two basic ways: byte by byte or line by line. The byte-by-byte format is most useful when you're reading binary data or when you're not going to be processing the information on a line-by-line basis. The line-by-line mode is best used when you expect to process the information on a line-by-line basis—say when processing a log file. The line-by-line mode has two separate modes: either you read a single line from the file or you read all the lines in the file at one time.

Line by Line

The **readline()** method reads a single line from the file:

```
line = file.readline()
```

Different platforms use different line-termination characters, and Python recognizes the appropriate line-termination character for your platform when reading a file. However, be warned that Python does not recognize the line-termination character for platforms other than the current platform. For example, you can read a Unix file line by line on a Unix system, but on a Mac, the same Unix file would appear as one big line to the Mac version of Python. Because Windows uses two characters for line termination, you can read individual lines from a file under Mac OS or Unix, but you'll have a spurious characters at the end or beginning of the lines you read from the file. See Chapter 23 for more information.

The line returned by the **readline** method includes the trailing line-termination characters, so you'll need to use the **string.rstrip()** method to take the new line off in many situations. For example, the temptation is to use a loop like the following to print a file to the screen:

```
    line = myfile.readline()
if line:
        print line
    else:
        break
```

The problem is that the output will include double line terminations, and therefore blank lines in the output. Instead, change the **print** line to

```
print line,
```

or use **sys.stdout.write()** to ensure that a line-termination character isn't automatically appended to the output.

Getting All the Lines

You can read all of the lines from a file using the **readlines()** method on an active file object. For example, you can read the entire file into a list using

```
lines = myfile.lines()
```

To read the entire contents of a file into a single string object, use the **read()** method without any arguments, as in the following example:

```
myfiledata = myfile.read()
```

Byte by Byte

You can also use the **read()** method to read a specific number of bytes from a file. For example, to read a 512-byte record from a file, use a statement like this:

```
record = file.read(512)
```

End of File

Python does not support the notion of an "end-of-file" status for an open file object. Although an **EOFError** exception does exist, it's actually used by the **input()** and **raw_input()** built-in functions to identify when an end-of-file character has been identified while reading input from the keyboard.

Instead, the **read()** and **readline()** methods return an empty string when they reach the end of the file. To end the processing of a file, you therefore need to check the information that you read from the file and break out of the loop accordingly:

```
while 1:
line = myfile.readline()
    if !line: break
    print line,
myfile.close()
```

The **if** test breaks out of the loop when it sees an empty line.

The other alternative is to use a **for** loop and the **readlines()** method to step through each individual line:

```
for line in myfile.readlines():
print line,
myfile.close()
```

This script automatically terminates when the list of lines returned by the **readlines()** call is reached.

Processing Example

Processing a file in Python requires more than just the basic ability to read lines or data. Most of the time you'll need to be able to identify different portions of the line or data that you read, and in fact, you saw many examples of this in Chapter 10. When working with data on a line-by-line basis, the obvious solution is to use **string.split()** to extract elements or to use the regular expression system offered by **re** to identify the elements you want.

If you are reading binary data, use the **struct** module to extract binary information. Alternatively, you may want to employ the services of the **pickle** module and similar modules described in Chapter 12 to store and load objects permanently in a file.

As an example of what you can do, here's a simple script in Python that uses **string.split()** to compile a count of the hosts and URL access from a standard web log:

```
import sys

def cmpval(tuple1, tuple2):
    return cmp(tuple2[1],tuple1[1])

hostaccess = {}
urlaccess = {}

if len(sys.argv) < 2:
    print "Usage:",sys.argv[0],"logfile"
    sys.exit(1)

try:
    file = open(sys.argv[1])
except:
    print "Whoa!","Couldn't open the file",sys.argv[1]
    sys.exit(1)

while 1:
    line = file.readline()
    if line:
        splitline = string.split(line)
        if len(splitline) < 10:
            print splitline
```

```
            continue
        (host,ident,user,time,offset,req,
         loc,httpver,success,bytes) = splitline
        try:
            hostaccess[host] = hostaccess[host] + 1
        except:
            hostaccess[host] = 1
        try:
            urlaccess[loc] = urlaccess[loc] + 1
        except:
            urlaccess[loc] = 1
    else:
        break

hosts = hostaccess.items()
hosts.sort(lambda f, s: cmp(s[1], f[1]))

for host, count in hosts:
    print host, ": ", count

urls = urlaccess.items()
urls.sort(cmpval)

for url, count in urls:
    print url, ": ", count
```


Writing to a File

Writing information to a file is usually just a case of calling the **write()** or **writelines()** methods.

Using write() or writelines()

The **write()** and **writelines()** methods are the most obvious way to write information to a file in Python. Both can be used to write binary data and unlike **print**, they do not automatically add a newline to each string written to the file. The **write()** method writes a single string, and despite the name, the **writelines()** method actually only writes a list of strings to the file. For example, to write a string to an open file, use this statement:

```
file.write('Some text')
```

You can write a list of strings using the following statement:

```
file.writelines(lines)
```

Using print

New in Python 2.0 is the ability to write directly to an open file without using the **write()** or **writelines()** methods. Instead, use a special operator that, in combination with the name of the file object that you want to write to, places the information between the **print** keyword and the string you want to write:

```
print >>file 'Some kind of error occurred'
```

Because you're using **print**, the newline character is automatically appended, so be careful when using **print** over **write()**—appending a comma stops Python from appending the line-termination sequence. Also be careful when supplying a list or dictionary to **print** because it writes a string representation of the list, tuple, or dictionary to the file, rather than emulating the **writelines()** method.

Changing Position

All file objects keep a record of their position within the file. This position is based on the number of bytes you have read or written to the file; a value 0 means you are writing at the start, before the first byte in the file. Position 1 defines the first byte of the file, position 2 defines the second byte, and position 512 defines the 512th byte in the file. If you add information to the file, it's added *after* your current position, so at position 1, you actually start writing information from byte 2 on. Note, of course, that whenever you write information to a file, you are *overwriting* the existing data, not inserting it.

You can determine your current position using the **tell()** method on an open file object. To change your position—outside of reading or writing the file of course—use the **seek()** method. The **seek()** method takes a single argument that specifies the location within the file in bytes that you want to move to. For example, to move to the start of the file, use

```
file.seek(0)
```

Moving to the end of the file is more difficult because you'd need to know the file's length in order to be able to move to the absolute position of the last byte in the file. There is a solution however. An optional second argument to the **seek()** method defines from where to take the location. The default is 0—all locations are assumed to be relative to the start of the file.

If the second argument to **seek()** is set to 1, the supplied first argument is taken as an addition to the current position. So, if you are at byte 512 and use the statement

```
file.seek(512,1)
```

you automatically move to byte 1024 (512+512). If the second argument is 2, the first argument is taken as relative to the end of the file. Therefore, you can move directly to the last byte in a file using the statement

```
file.seek(0,2)
```

You can also move back 512 bytes using the statement

```
file.seek(-512,2)
```

Table 11-4 summarizes these values.

Seek Value	Description
0	Relative to the start of the file (i.e., absolute). This is the default value.
1	Relative to the current position.
2	Relative to the end of the file.

Table 11-4. *Seek Position Values*

Controlling File I/O

Once you've opened a file, you may want to control how the file is accessed and shared. Under Unix you can do this using the **fcntl** module, which provides an interface to the underlying **fcntl.h** and **ioctl.h** header file and library interfaces in C.

File Control

To use file control, first get the file descriptor number from the file object you want to control. The **fileno** method does this for you.

Next, the **fcntl()** function in the **fcntl** module sets or gets the configuration information for the file. The basic format of the **fcntl()** function is

```
fnctl(fd, command [, args])
```

The **fd** argument is the file descriptor number returned by **fileno()**. The **command** argument is a constant that specifies the command you want to send to the file control system. Commands either set or get status and sharing information about the file. Table 11-5 contains a list of the valid commands and Table 11-6 contains a list of the supported options for the **F_SETFL** command. Note that the **FCNTL** module exports all of the constants in these two tables.

Command	Description
F_DUPFD	Duplicates a file descriptor. If **args** is supplied, it's used as the lowest possible number that should be used for the duplicate file descriptor; the actual value depends on the open file descriptors. The return value is the value of the new file descriptor.
F_SETFD	Sets the close-on-exec flag. If set to true (**args** is 1), the file descriptor is closed when an **exec*()** call is executed. If set to false (**args** is 0), the file descriptor remains open and is duplicated by the call (the default).
F_GETFD	Returns the current value of the close-on-exec flag.
F_SETFL	Sets the file status flag to **args**, which should be a bitwise OR of one or more of the constants defined in Table 11-6. Make sure to use the correct flag for your system type.
F_GETFL	Gets the file status flag.

Table 11-5. ***fcntl*** *Commands*

Command	Description
F_GETOWN	Gets the process ID or process group ID to receive the SIGIO signal (BSD only).
F_SETOWN	Sets the process ID to receive the SIGIO signal (BSD only).
F_GETLK	Gets the file lock structure. Not supported on all platforms.
F_SETLK	Locks a file. Returns –1 if the file is already locked. Not supported on all platforms.
F_SETLKW	Locks a file but blocks the execution of the current process until the lock can be acquired. Not supported on all platforms.

Table 11-5. *fcntl Commands* (continued)

I/O Control

The **ioctl()** function is identical to **fcntl()** except that it provides an interface to the **ioctl** subsystem for control I/O on any valid file descriptor. Check the documentation for your system to see what configurable options are supported.

File Locking

When working with any file, especially when writing, you may need to ensure that the current process is the only process that has the ability to write to the file. Although

System V	BSD	Description
O_NDELAY	FNDELAY	Non-blocking I/O— execution does not stop while reading or writing from the file. This shouldn't be necessary for most platforms.
O_APPEND	FAPPEND	Append mode.
O_SYNC	FASYNC	Synchronous I/O—no buffering is used so writing to the file automatically writes the information to disk. Under BSD this causes a SIGIO signal to be raised to the process group when I/O is possible.

Table 11-6. *Status Flag Constants for the F_SETFL fcntl Command*

Operation	Description
LOCK_EX	Exclusive lock. Other processes cannot even obtain a shared lock on the file.
LOCK_NB	Don't block the locking process. This causes the functions to immediately return with the status of the lock. If not specified, the processes block (wait) until the specified lock can be achieved.
LOCK_SH	Get a shared lock. This allows only your process to read the write to the file while allowing others to read.
LOCK_UN	Remove any lock.

Table 11-7. *File-Locking Options*

there are lots of potential solutions for this, the best way to handle the locking process is to use the operating system file-locking facilities. There are two functions for locking a file:

- **flock()** provides a simple whole-file locking mechanism.
- **lockf()** allows you to lock specific portions of a file.

Both functions support the same operations, which are listed in Table 11-7 and supported through the module by a number of constants defined in the **fcntl** module. Note that support for **fcntl** options is entirely OS dependent.

The **flock()** function is defined as follows:

```
flock(fd, op)
```

where **fd** is the file descriptor number and **op** is a bitwise OR of the file-locking options described in Table 11-7. The **fd** function locks the entire file until you unlock it or the process terminates.

The **lockf()** functions is defined as follows:

```
lockf(fd, op [, len [, start [, whence]]])
```

The **fd** and **op** arguments are the same as before but the **len**, **start**, and **whence** arguments define the length and duration of the lock. If not supplied with these arguments, the lock is set for the whole file. If **len** is specified, the **lockf()** function locks only the first **len** bytes of the file. The **start** and **whence** arguments work like the arguments to **seek** specifying the starting location and reference point of the lock. For example, you can lock the first 1,024 bytes with

```
fnctl.lockf(myfd, FCNTL.LOCK_EX, 1024)
```

or the last 1,024 bytes with

```
fnctl.lockf(myfd, FCNTL.LOCK_EX, 1024, -1024, 2)
```

Getting File Lists

Often you want to work with a list of files rather than a single file. For example, suppose you want to process all the **.html** files within the current directory. You can get around this by using the shell—under Unix, it is the shell (**sh**, **ksh**, **bash**, etc.) that actually expands a file specification such as ***.html** into a list of files that is then supplied directly to the application and available within Python as **sys.argv**.

If you want to get a list of files without using the directory, use the **glob** module. The **glob()** function within the **glob** module supports the same wildcard options as most Unix shells; Table 11-8 lists the supported wildcard specifications.

For example, to get a list of all HTML files, use

```
files = glob.glob('*.html')
```

The return value is a list of strings, with each element referring to a single file.

To perform more complex matches, use the regular expression module described in Chapter 10 to check each filename. For example, to find all the files with capital letters and a single number, for example, **THEFILE1.txt**, you might use something like this:

```
filelist = []
for file in glob.glob('./*'):
    if (re.search(r'[A-Z]*?[0-9].[a-z]*', file)):
        filelist.append(file)
```

Specification	Description
*	Matches anything and any quantity.
?	Matches a single character.
[seq]	Matches any character in **seq**.
[!seq]	Matches any character not in **seq**.

Table 11-8. *Wildcard Specifications for Finding Files*

Basic File/Directory Management

Most of the functions for moving, renaming, and otherwise manipulating directories can be found in the **os** module. Most of these functions are self-explanatory; they are summarized in Table 11-9.

Function	Description
chdir(name)	Changes the current working directory to **name**.
getcwd()	Returns the path to the current working directory.
link(source, dest)	Creates a hard link between **source** and **dest**. This function is supported by all platforms/operating systems.
mkdir(path [, mode])	Makes the directory **path** using the mode **mode**. Under Python, **mode** defaults to 0777 if not supplied.
makedirs(path [, mode])	Identical to **mkdir()** except that the **makedirs()** function makes all of the directories defined in the path, such that **makedirs('/a/b/c')** creates **'/a'**, **'/a/b'** and **'/a/b/c'** if they didn't already exist.
remove(path) or unlink(path)	Removes a given file.
removedirs(path)	Removes the directory specified in path, including any subdirectories or files (identical to **rm -r**). Raises an **OSError** exception if the directory cannot be removed.
rename(source, dest)	Renames **source** to **dest**.
renames(source, dest)	Identical to **rename()** except that any directories specified in **dest** that do not exist are created in an identical fashion to that used by **makedirs()**.
rmdir(path)	Removes the directory **path**.
symlink(source, dest)	Creates a symbolic link between **source** and **dest**. Not supported on all platforms/operating systems.

Table 11-9. *Making, Renaming, and Deleting Files/Directories in Python*

For more high-level options such as copying files and directory trees, use the **shutil** module. The **shutil** module includes functions for basic file copies, mode and permission copies, and utilities for duplicating and deleting entire directory trees. Table 11-10 contains the list of functions supported. Support is provided for all operating systems.

Function	Description
copyfile(src, dst)	Copies the file specified by the path **src** to **dst**.
copymode(src, dst)	Copies the permission bits for **src** to **dst**.
copystat(src, dst)	Copies the permissions, modification, and access times from **src** to **dst**. Note that this function does not copy the file contents nor does it change the owner or group for the file.
copy(src, dst)	Copies the file from **src** to the file or directory in **dst**.
copy2(src,dst)	Copies the file from **src** to the file or directory in **dst** and also copies the access and modification times of the files.
copytree(src, dst [,symlinks])	Copies the entire directory tree of the directory **src** to the directory **dst**. Files are copied using **copy2()**. If **symlinks** is true, symbolic links are recreated in the destination as symbolic links. If **symlinks** is false or unspecified, the file contents are copied as per a normal file.
rmtree(path [, ignore_errors [, onerror]])	Removes the entire directory tree in **path**. If **ignore_errors** is true, any errors are ignored. If **ignore_errors** is false, errors are handled by the function specified by **onerror**, which must accept three arguments: **func**, **path**, and **excinfo**. **func** refers to the function that caused the error (i.e., **rmdir()** or **remove()**), and **path** refers to the file or directory that was being deleted. **excinfo** is a set of exception information as raised by **sys.exc_info()**.

Table 11-10. *Functions for Copying and Removing Directories in shutil*

Access and Ownership

Often you need to determine or set the file options and permissions for a given file. The os module provides a series of functions to get and set file permissions and other information. You can also use the output in combination with the **stat** module to provide a cleaner interface to the information returned, or for some statistics, the **os.path** module.

Checking Access

To check your ability to access a file, use the **os.access()** function:

```
os.access(path, accessmode)
```

path should be the path to the file or directory you want to check, and **accessmode** is a constant that specifies what sort of access you want to check. Valid values for **accessmode** are:

- **R_OK** (able to read)
- **W_OK** (able to write)
- **X_OK** (able to execute)
- **F_OK** (to test for existence)

For example, to check whether you have permission to read a file, use:

```
os.access('myfile.txt', os.R_OK)
```

Getting File Information

To get information about a file (i.e., its permissions, ownership, and access times), use the **os.stat** function (for files) or the **os.lstat** function (for links). Both functions return the same information in the form of a tuple of values containing the information for a file. Table 11-11 lists the values returned and their element reference along with the constant exported by the **stat** module for accessing the information by name rather than index.

Element Reference	stat Constant	Description
0	ST_MODE	File mode (type and permissions)
1	ST_INO	Inode number
2	ST_DEV	Device number of file system
3	ST_NLINK	Number of (hard) links to the file
4	ST_UID	Numeric user ID of file's owner

Table 11-11. *Elements Returned by* ***stat()***

Element Reference	stat Constant	Description
5	ST_GID	Numeric group ID of file's owner
6	ST_SIZE	File size, in bytes
7	ST_ATIME	Last access time since the epoch
8	ST_MTIME	Last modify time since the epoch
9	ST_CTIME	Inode change time (*not* creation time!) since the epoch

Table 11-11. *Elements Returned by* ***stat()*** (continued)

For example, to get the size of a file, use:

```
filesize = os.stat(file)[6]
```

You can use the output of the **stat()** commands with the **stat** module in order to get, set, or determine the file type and other information. The **stat** module exports constants to access the different elements of a **stat()** tuple, as shown in Table 11-11 in the **stat** Constant column. In addition, it provides the functions in Table 11-12, which return summary information for a given aspect of the file. Note that all the functions accept a single argument: the mode as returned by **os.stat(path)[stat.ST_MODE]**.

Function	Description
S_ISDIR(mode)	Returns true if the path is a directory.
S_ISCHR(mode)	Returns true if the file is a character special device file.
S_ISBLK(mode)	Returns true if the file is a block special device file.
S_ISREG(mode)	Returns true if the file is a regular file.
S_ISFIFO(mode)	Returns true if the path is a FIFO.
S_ISLNK(mode)	Returns true if the path is a symbolic link.
S_ISSOCK(mode)	Returns true if the path is a Unix socket.
S_IMODE(mode)	Returns the portion of the file's mode that can be set by the **os.chmod()** function.
S_IFMT(mode)	Returns the portion of the file's mode that describes the file's type.

Table 11-12. *Test Functions Exported by the* ***stat*** *Module*

A quicker interface is available through the **os.path** module, which provides utility functions to obtain most of this information with a single function call. Table 11-13 lists the supported functions in **os.path** for determining file information.

Setting File Permissions

To actually set the file mode and ownership within Python, use the **os.chmod()** and **os.chown()** functions respectively. Note that these functions only apply to Unix-style operating systems (including QNX, BeOS, and MacOS X), but not to Windows or MacOS. For example, to set the permissions on a file, use

```
os.chmod('file.txt', 0666)
```

Note that the permissions must be specified as an octal number, just as you would when specifying the mode on the command line in Unix. The **os.chown()** function accepts three arguments: the path, the user ID, and the group ID. For example, to change **file.txt** to be owned by user 1001 and group 1000, use the following statement:

```
os.chown('file.txt', 1001, 1000)
```

To use names rather than numbers, use the **pwd** and **grp** modules to determine the numeric IDs for a given user or group.

For cross-reference purposes, Table 11-14 contains the format of the **os.chmod()** and **os.chown()** functions.

Function	Description
getatime(path)	Returns the time of last access as the number of seconds since the epoch.
getmtime(path)	Returns the time of last modification as the number of seconds since the epoch.
getsize(path)	Returns the size of the file in bytes.

Table 11-13. *Alternative Methods for Getting File Information in Python*

Python Function	Description
os.chmod(path, mode)	Change the permission mode of path to mode.
os.chown(path, uid, gid)	Change the user ID and group ID of path.

Table 11-14. *Setting Access Modes in Python*

Manipulating File Paths

When working with files, you'll inevitably need to manipulate the directories and files or the paths of the files and directories that you want to open. Manipulating this information is not easy, especially if you want to support different platforms. For example, the Unix platform uses the forward slash (/) character to separate directory components, but MacOS uses a colon (:) and officially Windows uses the backslash (\) character, although the forward slash (/) character is supported on Windows 95 and later versions.

To make the process easier, the **os** module exports variables that define the settings for the current platform. Table 11-15 lists the variables that hold this information.

Variable	Description
sep	The character used to separate path components: / on Unix, : on MacOS, and \ on Windows.
pathsep	The character used to separate each path within the $PATH environment variable: : for Unix and ; for Windows/DOS.
pardir	The character sequence used to describe the parent directory: .. on Unix/Windows and :: on MacOS.
curdir	The character sequence used to describe the current directory: . on Unix and Windows and : on MacOS.
altsep	The character used as an alternative separator. This really only applies to Windows where the forward slash (/) can be used.

Table 11-15. *Directory Component Variables in the* **os** *Module*

Armed with this information, you can assemble a file's path using

```
fullpath = basedir + os.sep + dir + os.sep + file
```

However, this is messy and it doesn't actually account for such niceties as removing duplicate separators in the final path. A much better solution is offered by the **os.path** module, which provides a number of functions to assemble and disassemble a given path on any platform into the correct format. Table 11-16 lists these functions.

The **basename()** and **dirname()** functions both disassemble a path and return the directory and/or filename from a path, as in the following example:

```
>>> os.path.basename('/usr/local/lib/python2.1/os.py')
'os.py'
>>> os.path.dirname('/usr/local/lib/python2.1/os.py')
'/usr/local/lib/python2.1'
```

The **abspath()** function cleans up a path, resolving references to current and parent directories in order to arrive at a final, cleaned path:

```
'/var/adm/su.log'
```

Finally, the **join()** function joins together components, including the correct separators for the current platform and returns a clean and valid path:

```
'/usr/local/python2.1/os.py'
```

Python Function	Description
os.path.join(patha [, pathb ...])	Joins the components **patha**, **pathb** into a valid path using the correct path separators for the current platform.
os.path.abspath(path)	Returns a cleaned-up version of **path**, removing references to current and parent directories.
os.path.basename(path)	Extract the base name (filename or final directory) for a given path.
os.path.dirname(path)	Extract the directory name for a given path.

Table 11-16. *Path Manipulation Functions in Python*