Tutorial on File and Directory Access in Python

Norman Matloff
University of California, Davis

August 7, 2007

Contents

1 Files

1.1 Some Basic File Operations

A list of Python file operations can be obtained through the Python **dir()** command:

```
>>> dir(file)
['__class__', '__delattr__', '__doc__', '__getattribute__', '__hash__',
'__init__', '__iter__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__str__', 'close', 'closed', 'encoding',
'fileno', 'flush', 'isatty', 'mode', 'name', 'newlines', 'next', 'read',
'readinto', 'readline', 'readlines', 'seek', 'softspace', 'tell',
'truncate', 'write', 'writelines', 'xreadlines']
```

Following is an overview of many of those operations. I am beginning in a directory **/a**, which has a file **x**, consisting of

```
a
bc
def
```

a file **y**, consisting of

```
uuu
vvv
```

and a subdirectory **b**, which is empty.

```
1  >>> f = open('x')  # open, read-only (default)
2  >>> f.name  # recover name from file object
3  'x'
4  >>> f.readlines()
5  ['a\n', 'bc\n', 'def\n']
6  >>> f.readlines()  # already at end of file, so this gets nothing
7  []
8  >>> f.seek(0)  # go back to byte 0 of the file
9  >>> f.readlines()
10 ['a\n', 'bc\n', 'def\n']
11 >>> f.seek(2)  # go to byte 2 in the file
12 >>> f.readlines()
13 ['bc\n', 'def\n']
14 >>> f.seek(0)
15 >>> f.read()  # read to EOF, returning bytes in a string
16 'a\nbc\ndef\n'
17 >>> f.seek(0)
18 >>> f.read(3)  # read only 3 bytes this time
19 'a\nb'
```

If you need to read just one line, use **readline()**.

Starting with Python 2.3, you can use a file as an iterator:

```
1  >>> f = open('x')
2  >>> i = 0
3  >>> for l in f:
```

```
...     print 'line %d:' % i,l[:-1]
...     i += 1
...
line 0: a
line 1: bc
line 2: def
```

Note, by the way, how we stripped off the newline character (which we wanted to do because **print** would add one and we don't want two) by using **l[:-1]** instead of **l**.

So, how do we write to files?

```
>>> g = open('z','w')  # open for writing
>>> g.write('It is a nice sunny day')  # takes just one argument
>>> g.write(' today.\n')
>>> g.close()
>>> import os
>>> os.system('cat z')  # issue a shell command, which also gives an exit code
It is a nice sunny day today.
0
>>> g = open('gy','w')
>>> g.writelines(['Here is a line,\n','and another.\n'])
>>> os.system('cat gy')
0
>>> g.close()
>>> os.system('cat gy')
Here is a line,
and another.
0
>>> a = open('yy','w')
>>> a.write('abc\n')
>>> a.close()
>>> os.system('cat yy')
abc
0
>>> b = open('yy','a')  # open in append mode
>>> b.write('de\n')
>>> b.close()
>>> os.system('cat yy')
abc
de
0
```

As in Unix, **stdin** and **stdout** count as files too (*file-like objects*, in Python parlance), so we can use the same operations, e.g.:

```
>>> import sys
>>> w = sys.stdin.readlines()  # keyboard input, ctrl-d to end
moon, sun
and
stars
>>> w
['moon, sun\n', 'and\n', 'stars\n']
```

2 Directories

2.1 Some Basic Directory Operations

The **os** module is huge. Here's a start to seeing how big it is:

```
>>> import os
>>> dir(os)
['EX_CANTCREAT', 'EX_CONFIG', 'EX_DATAERR', 'EX_IOERR', 'EX_NOHOST',
'EX_NOINPUT', 'EX_NOPERM', 'EX_NOUSER', 'EX_OK', 'EX_OSERR',
'EX_OSFILE', 'EX_PROTOCOL', 'EX_SOFTWARE', 'EX_TEMPFAIL',
'EX_UNAVAILABLE', 'EX_USAGE', 'F_OK', 'NGROUPS_MAX', 'O_APPEND',
'O_CREAT', 'O_DIRECT', 'O_DIRECTORY', 'O_DSYNC', 'O_EXCL',
'O_LARGEFILE', 'O_NDELAY', 'O_NOCTTY', 'O_NOFOLLOW', 'O_NONBLOCK',
'O_RDONLY', 'O_RDWR', 'O_RSYNC', 'O_SYNC', 'O_TRUNC', 'O_WRONLY',
'P_NOWAIT', 'P_NOWAITO', 'P_WAIT', 'R_OK', 'TMP_MAX', 'UserDict',
'WCOREDUMP', 'WEXITSTATUS', 'WIFEXITED', 'WIFSIGNALED', 'WIFSTOPPED',
'WNOHANG', 'WSTOPSIG', 'WTERMSIG', 'WUNTRACED', 'W_OK', 'X_OK',
'_Environ', '__all__', '__builtins__', '__doc__', '__file__',
'__name__', '_copy_reg', '_execvpe', '_exists', '_exit',
'_get_exports_list', '_make_stat_result', '_make_statvfs_result',
'_pickle_stat_result', '_pickle_statvfs_result', '_spawnvef',
'_urandomfd', 'abort', 'access', 'altsep', 'chdir', 'chmod', 'chown',
'chroot', 'close', 'confstr', 'confstr_names', 'ctermid', 'curdir',
'defpath', 'devnull', 'dup', 'dup2', 'environ', 'error', 'execl',
'execle', 'execlp', 'execlpe', 'execv', 'execve', 'execvp', 'execvpe',
'extsep', 'fchdir', 'fdatasync', 'fdopen', 'fork', 'forkpty',
'fpathconf', 'fstat', 'fstatvfs', 'fsync', 'ftruncate', 'getcwd',
'getcwdu', 'getegid', 'getenv', 'geteuid', 'getgid', 'getgroups',
'getloadavg', 'getlogin', 'getpgid', 'getpgrp', 'getpid', 'getppid',
'getsid', 'getuid', 'isatty', 'kill', 'killpg', 'lchown', 'linesep',
'link', 'listdir', 'lseek', 'lstat', 'major', 'makedev', 'makedirs',
'minor', 'mkdir', 'mkfifo', 'mknod', 'name', 'nice', 'open', 'openpty',
'pardir', 'path', 'pathconf', 'pathconf_names', 'pathsep', 'pipe',
'popen', 'popen2', 'popen3', 'popen4', 'putenv', 'read', 'readlink',
'remove', 'removedirs', 'rename', 'renames', 'rmdir', 'sep', 'setegid',
'seteuid', 'setgid', 'setgroups', 'setpgid', 'setpgrp', 'setregid',
'setreuid', 'setsid', 'setuid', 'spawnl', 'spawnle', 'spawnlp',
'spawnlpe', 'spawnv', 'spawnve', 'spawnvp', 'spawnvpe', 'stat',
'stat_float_times', 'stat_result', 'statvfs', 'statvfs_result',
'strerror', 'symlink', 'sys', 'sysconf', 'sysconf_names', 'system',
'tcgetpgrp', 'tcsetpgrp', 'tempnam', 'times', 'tmpfile', 'tmpnam',
'ttyname', 'umask', 'uname', 'unlink', 'unsetenv', 'urandom', 'utime',
'wait', 'waitpid', 'walk', 'write']
```

To see some examples, we start in the directory **/a** as above.

```
1  >>> os.path.curdir  # get current directory (Pythonic humor)
2  '.'
3  >>> t = os.path.abspath(os.path.curdir)  # get its full path
4  >>> t
5  '/a'
6  >>> os.path.basename(t)  # get the tail end of the path
7  'a'
8  >>> os.getcwd()  # another way to get the full path
9  '/a'
10 >>> cwd = os.getcwd()
11 >>> os.listdir(cwd)  # see what's in this directory (excl. '.','..')
12 ['b', 'x', 'y', 'z']
13 >>> os.stat('x')  # get all the information about a file
14 (33152, 5079544L, 773L, 1, 0, 0, 9L, 1112979025, 1112977272, 1112977272)
15 >>> # the numbers above are: inode protection mode; inode number; device
16 >>> # inode resides on; number of links to the inode; user id of the owner;
17 >>> # group id of the owner; size in bytes; time of last access; time
18 >>> # of last modification; "ctime" as reported by OS
19 >>> os.path.getsize('x')  # get file size; argument must be string
20 9L
21 >>> os.path.isdir('x')  # is this file a directory?
22 False
23 >>> os.path.isdir('b')
24 True
25 >>> os.path.getmtime('x')  # time of last modification (seconds since Epoch)
```

```
1112977272
>>> import time
>>> time.ctime(os.path.getmtime('x'))  # translate that to English
'Fri Apr  8 09:21:12 2005'
>>> os.chdir('b')  # change to subdirectory b
>>> os.listdir('.')
[]
>>> os.mkdir('c')  # make a new directory
>>> os.listdir('.')
['c']
```

2.2 The Powerful `walk()` Function

The function **os.walk()** does a recursive descent down a directory tree, stopping in each subdirectory to perform user-coded actions. This is quite a powerful tool.[1]

The form of the call is

```
os.path.walk(rootdir,f,arg)
```

where **rootdir** is the name of the root of the desired directory tree, **f()** is a user-supplied function, and **arg** will be one of the arguments to **f()**, as explained below.

At each directory **d** visited in the "walk," **walk()** will call **f()**, and will provide **f()** with the list **flst** of the names of the files in **d**. In other words, **walk()** will make this call:

```
f(arg,d,flst)
```

So, the user must write **f()** to perform whatever operations she needs for the given directory. Remember, the user sets **arg** too. According to the Python help file for **walk()**, in many applications the user sets **arg** to **None** (though not in our example here).

The following example is adapted from code written by Leston Buell[2], which found the total number of bytes in a directory tree. The differences are: Buell used classes to maintain the data, while I used a list; I've added a check for linked files; and I've added code to also calculate the number of files and directories (the latter counting the root directory).

```
# walkex.py; finds the total number of bytes, number of files and number
# of directories in a given directory tree, dtree (current directory if
# not specified); adapted from code by Leston Buell

# usage:
#    python walkex.py [dtree_root]

import os, sys

def getlocaldata(sms,dr,flst):
   for f in flst:
      # get full path name relative to where program is run; the
      # function os.path.join() adds the proper delimiter for the OS,
      #    e.g. / for Unix, \ for Windows
      fullf = os.path.join(dr,f)
```

[1] Unix users will recognize some similarity to the Unix **find** command.

[2] `http://fizzylogic.com/users/bulbul/programming/dirsize.py`

```
16          if os.path.islink(fullf): break  # don't count linked files
17          if os.path.isfile(fullf):
18             sms[0] += os.path.getsize(fullf)
19             sms[1] += 1
20          else:
21             sms[2] += 1
22
23    def dtstat(dtroot):
24       sums = [0,0,1]  # 0 bytes, 0 files, 1 directory so far
25       os.path.walk(dtroot,getlocaldata,sums)
26       return sums
27
28    def main():
29       try:
30          root = sys.argv[1]
31       except:
32          root = '.'
33       report = dtstat(root)
34       print report
35
36    if __name__== '__main__':
37       main()
```

Important feature: When **walk()** calls the user-supplied function **f()**, transmitting the list **flist** of files in the currently-visited directory, **f()** may modify that list. The key point is that **walk()** will continue to use that list to find more directories to visit.

For example, suppose there is a subdirectory **qqq** in the currently-visited directory. The function **f()** could delete **qqq** from **flist**, with the result being that **walk()** will NOT visit the subtree having **qqq** as its root.[3]

3 Cross-Platform Issues

Like most scripting languages, Python is nominally cross-platform, usable on Unix, Windows and Macs. It makes a very serious attempt to meet this goal, and succeeds reasonably well. Let's take a closer look at this.

3.1 The Small Stuff

Some aspects cross-platform compatibility are easy to deal with. One example of this is file path naming, e.g. **/a/b/c** in Unix and **\a\b\c** in Windows. We saw above that the library functions such as **os.path.abspath()** will place slashes or backslashes according to our underlying OS, thus enabling platform-independent code. In fact, the quantity **os.sep** stores the relevant character, '/' for Unix and '\' for Windows.

Similarly, in Unix, the end-of-line marker (EOL) is a single byte, 0xa, while for Windows it is a pair of bytes, 0xd and 0xa. The EOL marker is stored in **os.linesep**.

3.2 How Is It Done?

Let's take a look at the **os** module, which will be in your file **/usr/lib/python2.4/os.py** or something similar. You will see code like

[3] Of course, if it has already visited that subtree, that can't be undone.

```
_names = sys.builtin_module_names
...
if 'posix' in _names:
    name = 'posix'
    linesep = '\n'
    from posix import *
    try:
        from posix import _exit
    except ImportError:
        pass
    import posixpath as path

    import posix
    __all__.extend(_get_exports_list(posix))
    del posix
```

(POSIX is the name of "standard" Unix.)

3.3 Python and So-Called "Binary" Files

Keep in mind that the term **binary file** is a misnomer. After all, ANY file is "binary," whether it consists of "text" or not, in the sense that it consists of bits, no matter what. So what do people mean when they refer to a "binary" file?

First, let's define the term **text file** to mean a file satisifying all of the following conditions:

(a) Each byte in the file is in the ASCII range 00000000-01111111, i.e. 0-127.

(b) Each byte in the file is *intended* to be thought of as an ASII character.

(c) The file is *intended* to be broken into what we think of (and typically display) as lines. Here the term *line* is defined technically in terms of end-of-line markers.

Any file which does not satisfy the above conditions has traditionally been termed a **binary file**.[4]

The default mode of Python in reading a file is to assume it is a text file. Whenever an EOL marker is encountered, it will be converted to '\n', i.e. 0xa. This would of course be no problem on Unix platforms, since it would involve no change at all, but under Windows, if the file were actually a binary file and simply by coincidence contained some 0xda pairs, one byte from each pair would be lost, with potentially disastrous consequences. So, you must warn the system if it is a binary file. For example,

```
f = open('yyy','rb')
```

would open the file **yyy** in read-only mode, and treat the file as binary.

[4]Even this definition is arguably too restrictive. If we produce a non-English file which we intend as "text," it will have some non-ASCII bytes.