Working With Files And Directories

WE'LL COVER THE FOLLOWING

Python 3 comes with a module called os, which stands for “operating system.” The os module contains a plethora of functions to get information on — and in some cases, to manipulate — local directories, files, processes, and environment variables. Python does its best to offer a unified api across all supported operating systems so your programs can run on any computer with as little platform-specific code as possible.

The Current Working Directory

When you’re just getting started with Python, you’re going to spend a lot of time in the Python Shell. Throughout this book, you will see examples that go like this:

1. Import one of the modules in the examples folder
2. Call a function in that module
3. Explain the result

There is always a current working directory.

If you don’t know about the current working directory, step 1 will probably fail with an `ImportError`. Why? Because Python will look for the `example` module in the import search path, but it won't find it because the `examples`

folder isn't one of the directories in the search path. To get past this, you can do one of two things:

1. Add the `examples` folder to the import search path
2. Change the current working directory to the `examples` folder

The current working directory is an invisible property that Python holds in memory at all times. There is always a current working directory, whether you're in the Python Shell, running your own Python script from the command line, or running a Python cgi script on a web server somewhere.

The `os` module contains two functions to deal with the current working directory.

```
import os, sys                                         #①
os.getcwd()                                            #②
#/C:\Python31

os.chdir('/Users/pilgrim/diveintopython3/examples')    #③
os.getcwd()                                            #④
#C:\Users\pilgrim\diveintopython3\examples
```

① The `os` module comes with Python; you can import it anytime, anywhere.

② Use the `os.getcwd()` function to get the current working directory. When you run the graphical Python Shell, the current working directory starts as the directory where the Python Shell executable is. On Windows, this depends on where you installed Python; the default directory is `c:\Python31`. If you run the Python Shell from the command line, the current working directory starts as the directory you were in when you ran python3.

③ Use the `os.chdir()` function to change the current working directory.

④ When I called the `os.chdir()` function, I used a Linux-style pathname (forward slashes, no drive letter) even though I'm on Windows. This is one of the places where Python tries to paper over the differences between operating systems.

Working With Filenames and Directory Names

While we're on the subject of directories, I want to point out the os.path module. `os.path` contains functions for manipulating filenames and directory names.

```
import os
print(os.path.join('/Users/pilgrim/diveintopython3/examples/', 'humansize.py'))
#/Users/pilgrim/diveintopython3/examples/humansize.py

print(os.path.join('/Users/pilgrim/diveintopython3/examples', 'humansize.py'))
#/Users/pilgrim/diveintopython3/examples\humansize.py

print(os.path.expanduser('~'))
#/nonexistent

print(os.path.join(os.path.expanduser('~'), 'diveintopython3', 'examples', 'humansize.py'))
#/nonexistent/diveintopython3/examples/humansize.py
```

① The `os.path.join()` function constructs a pathname out of one or more partial pathnames. In this case, it simply concatenates strings.

② In this slightly less trivial case, calling the `os.path.join()` function will add an extra slash to the pathname before joining it to the filename. It's a backslash instead of a forward slash, because I constructed this example on Windows. If you replicate this example on Linux or Mac OS X, you'll see a forward slash instead. Don't fuss with slashes; always use `os.path.join()` and let Python do the right thing.

③ The `os.path.expanduser()` function will expand a pathname that uses `~` to represent the current user's home directory. This works on any platform where users have a home directory, including Linux, Mac OS X, and Windows. The returned path does not have a trailing slash, but the `os.path.join()` function doesn't mind.

④ Combining these techniques, you can easily construct pathnames for directories and files in the user's home directory. The `os.path.join()` function can take any number of arguments. I was overjoyed when I discovered this, since `addSlashIfNecessary()` is one of the stupid little functions I always need to write when building up my toolbox in a new language. Do not write this stupid little function in Python; smart people have already taken care of it for you.

`os.path` also contains functions to split full pathnames, directory names, and filenames into their constituent parts.

```
import os
pathname = '/Users/pilgrim/diveintopython3/examples/humansize.py'
print (os.path.split(pathname) )                                       #①
#('/Users/pilgrim/diveintopython3/examples', 'humansize.py')

(dirname, filename) = os.path.split(pathname)                          #②
print (dirname )                                                       #③
#/Users/pilgrim/diveintopython3/examples

print (filename )                                                      #④
#humansize.py

(shortname, extension) = os.path.splitext(filename)                    #⑤
print (shortname)
#humansize

print (extension)
#.py
```

① The `split` function splits a full pathname and returns a tuple containing the path and filename.

② Remember when I said you could use multi-variable assignment to return multiple values from a function? The `os.path.split()` function does exactly that. You assign the return value of the `split` function into a tuple of two variables. Each variable receives the value of the corresponding element of the returned tuple.

③ The first variable, `dirname`, receives the value of the first element of the tuple returned from the `os.path.split()` function, the file path.

④ The second variable, `filename`, receives the value of the second element of the tuple returned from the `os.path.split()` function, the filename.

⑤ `os.path` also contains the `os.path.splitext()` function, which splits a filename and returns a tuple containing the filename and the file extension. You use the same technique to assign each of them to separate variables.

Listing Directories

The `glob` module is another tool in the Python standard library. It's an easy way to get the contents of a directory programmatically, and it uses the sort of wildcards that you may already be familiar with from working on the command line.

The glob module uses shell-like wildcards.

```
import os
os.chdir('/Users/pilgrim/diveintopython3/')
import glob

glob.glob('examples/*.xml')                  #①
#['examples\\feed-broken.xml',
# 'examples\\feed-ns0.xml',
# 'examples\\feed.xml']

os.chdir('examples/')                        #②
glob.glob('*test*.py')                       #③
#['alphameticstest.py',
# 'pluraltest1.py',
# 'pluraltest2.py',
# 'pluraltest3.py',
# 'pluraltest4.py',
# 'pluraltest5.py',
# 'pluraltest6.py',
# 'romantest1.py',
# 'romantest10.py',
# 'romantest2.py',
# 'romantest3.py',
# 'romantest4.py',
# 'romantest5.py',
# 'romantest6.py',
# 'romantest7.py',
# 'romantest8.py',
# 'romantest9.py']
```

① The glob module takes a wildcard and returns the path of all files and directories matching the wildcard. In this example, the `wildcard` is a directory path plus "*.xml", which will match all `.xml` files in the `examples` subdirectory.

② Now change the current working directory to the `examples` subdirectory. The `os.chdir()` function can take relative pathnames.

③ You can include multiple wildcards in your `glob` pattern. This example finds all the files in the current working directory that end in a `.py` extension and contain the word `test` anywhere in their filename.

Getting File Metadata

Getting File Metadata

Every modern file system stores metadata about each file: creation date, last-modified date, file size, and so on. Python provides a single `API` to access this metadata. You don’t need to open the file; all you need is the filename.

```
import os
print(os.getcwd())                          #①
#/usercode

metadata = os.stat('feed.xml')              #②
print (metadata.st_mtime)                   #③
#1317786276.0

import time                                 #④
print (time.localtime(metadata.st_mtime))   #⑤
#time.struct_time(tm_year=2011, tm_mon=10,
# tm_mday=5, tm_hour=3, tm_min=44, tm_sec=36, tm_wday=2, tm_yday=278, tm_isdst=0)
```

① The current working directory is the `examples` folder.

② `feed.xml` is a file in the `examples` folder. Calling the `os.stat()` function returns an object that contains several different types of metadata about the file.

③ `st_mtime` is the modification time, but it’s in a format that isn’t terribly useful. (Technically, it’s the number of seconds since the Epoch, which is defined as the first second of January 1st, 1970. Seriously.)

④ The time module is part of the Python standard library. It contains functions to convert between different time representations, format time values into strings, and fiddle with timezones.

⑤ The `time.localtime()` function converts a time value from seconds-since-the-Epoch (from the `st_mtime` property returned from the `os.stat()` function) into a more useful structure of year, month, day, hour, minute, second, and so on. This file was last modified on July 13, 2009, at around 5:25 PM.

```
# continued from the previous example

metadata.st_size                                #①
#3070
```

```
import humansize


humansize.approximate_size(metadata.st_size)  #②
#'3.0 KiB'
```

① The `os.stat()` function also returns the size of a file, in the `st_size` property. The file `feed.xml` is `3070` bytes.

② You can pass the `st_size` property to the `approximate_size()` function.

Constructing Absolute Pathnames

In the previous section, the `glob.glob()` function returned a list of relative pathnames. The first example had pathnames like `'examples\feed.xml'`, and the second example had even shorter relative pathnames like `'romantest1.py'`. As long as you stay in the same current working directory, these relative pathnames will work for opening files or getting file metadata. But if you want to construct an absolute pathname — i.e. one that includes all the directory names back to the root directory or drive letter — then you'll need the `os.path.realpath(`) function.

```
import os
print(os.getcwd())
#/usercode

print(os.path.realpath('feed.xml'))
#/usercode/feed.xml
```