Chapter 16 Automating common tasks on your computer

We have been reading data from files, networks, services, and databases. Python can also go through all of the directories and folders on your computers and read those files as well.

In this chapter, we will write programs that scan scan through your computer and perform some operation on each file. Files are organized into directories (also called "folders"). Simple Python scripts can make short work of simple tasks that must be done to hundreds or thousands of files spread across a directory tree or your entire computer.

To walk through all the directories and files in a tree we use `os.walk` and a `for` loop. This is similar to how `open` allows us to write a loop to read the contents of a file, `socket` allows us to write a loop to read the contents of a network connection, and `urllib` allows us to open a web document and loop through its contents.

16.1 File names and paths

Every running program has a "current directory," which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.

The `os` module provides functions for working with files and directories (`os` stands for "operating system"). `os.getcwd` returns the name of the current directory:

```
>>> import os
>>> cwd = os.getcwd()
>>> print cwd
/Users/csev
```

`cwd` stands for **current working directory**. The result in this example is `/Users/csev`, which is the home directory of a user named `csev`.

A string like `cwd` that identifies a file is called a path. A **relative path** starts from the current directory; an **absolute path** starts from the topmost directory in the file system.

The paths we have seen so far are simple file names, so they are relative to the current directory. To find the absolute path to a file, you can use `os.path.abspath`:

```
>>> os.path.abspath('memo.txt')
'/Users/csev/memo.txt'
```

`os.path.exists` checks whether a file or directory exists:

```
>>> os.path.exists('memo.txt')
True
```

If it exists, `os.path.isdir` checks whether it's a directory:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

Similarly, `os.path.isfile` checks whether it's a file.

`os.listdir` returns a list of the files (and other directories) in the given directory:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

16.2 Example: Cleaning up a photo directory

Some time ago, I built a bit of Flickr-like software that received photos from my cell phone and stored those photos on my server. I wrote this before Flickr existed and kept using it after Flickr existed because I wanted to keep original copies of my images forever.

I would also send a simple one-line text description in the MMS message or the subject line of the e-mail message. I stored these messages in a text file in the same directory as the image file. I came up with a directory structure based on the month, year, day and time the photo was taken. The following would be an example of the naming for one photo and its existing description:

```
./2006/03/24-03-06_2018002.jpg
./2006/03/24-03-06_2018002.txt
```

After seven years, I had a lot of photos and captions. Over the years as I switched cell phones, sometimes my code to extract the caption from the message would break and add a bunch of useless data on my server instead of a caption.

I wanted to go through these files and figure out which of the text files were really captions and which were junk and then delete the bad files. The first thing to do was to get a simple inventory of how many text files I had in of the sub-folders using the following program:

```
import os
count = 0
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           count = count + 1
print 'Files:', count

python txtcount.py
Files: 1917
```

The key bit of code that makes this possible is the `os.walk` library in Python. When we call `os.walk` and give it a starting directory, it will "walk" through all of the directories and sub-directories recursively. The string "." indicates to start in the current directory and walk downward. As it encounters each directory, we get three values in a tuple in the body of the `for` loop. The first value is the current directory name, the second value is the list of sub-directories in the current directory, and the third value is a list of files in the current directory.

We do not have to explicitly look into each of the sub-directories because we can count on `os.walk` to visit every folder eventually. But we do want to look at each file, so we write a simple `for` loop to examine each of the files in the current directory. We check each file to see if it ends with ".txt" and then count the number of files through the whole directory tree that end with the suffix ".txt".

Once we have a sense of how many files end with ".txt", the next thing to do is try to automatically determine in Python which files are bad and which files are good. So we write a simple program to print out the files and the size of each file:

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           thefile = os.path.join(dirname,filename)
           print os.path.getsize(thefile), thefile
```

Now instead of just counting the files, we create a file name by concatenating the directory name with the name of the file within the directory using `os.path.join`. It is important to use `os.path.join` instead of string concatenation because on Windows we use a backslash (\) to construct file paths and on Linux or Apple we use a forward slash (/) to construct file paths. The `os.path.join` knows these differences and knows what system we are running on and it does the proper concatenation depending on the system. So the same Python code runs on either Windows or Unix-style systems.

Once we have the full file name with directory path, we use the `os.path.getsize` utility to get the size and print it out, producing the following output:

```
python txtsize.py
...
18 ./2006/03/24-03-06_2303002.txt
22 ./2006/03/25-03-06_1340001.txt
22 ./2006/03/25-03-06_2034001.txt
...
2565 ./2005/09/28-09-05_1043004.txt
2565 ./2005/09/28-09-05_1141002.txt
...
2578 ./2006/03/27-03-06_1618001.txt
2578 ./2006/03/28-03-06_2109001.txt
2578 ./2006/03/29-03-06_1355001.txt
...
```

Scanning the output, we notice that some files are pretty short and a lot of the files are pretty large and the same size (2578 and 2565). When we take a look at a few of these larger files by hand, it looks like the large files are nothing but a generic bit of identical HTML that came in from mail sent to my system from my T-Mobile phone:

```
<html>
        <head>
                <title>T-Mobile</title>
...
```

Skimming through the file, it looks like there is no good information in these files so we can probably delete them.

But before we delete the files, we will write a program to look for files that are more than one line long and show the contents of the file. We will not bother showing ourselves those files that are exactly 2578 or 2565 characters long since we know that these files have no useful information.

So we write the following program:

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           thefile = os.path.join(dirname,filename)
           size = os.path.getsize(thefile)
           if size == 2578 or size == 2565:
               continue
           fhand = open(thefile,'r')
           lines = list()
           for line in fhand:
               lines.append(line)
           fhand.close()
           if len(lines) > 1:
                print len(lines), thefile
                print lines[:4]
```

We use a `continue` to skip files with the two "bad sizes", then open the rest of the files and read the lines of the file into a Python list and if the file has more than one line we print out how many lines are in the file and print out the first three lines.

It looks like filtering out those two bad file sizes, and assuming that all one-line files are correct, we are down to some pretty clean data:

```
python txtcheck.py
3 ./2004/03/22-03-04_2015.txt
['Little horse rider\r\n', '\r\n', '\r']
2 ./2004/11/30-11-04_1834001.txt
['Testing 123.\n', '\n']
3 ./2007/09/15-09-07_074202_03.txt
['\r\n', '\r\n', 'Sent from my iPhone\r\n']
3 ./2007/09/19-09-07_124857_01.txt
['\r\n', '\r\n', 'Sent from my iPhone\r\n']
3 ./2007/09/20-09-07_115617_01.txt
...
```

But there is one more annoying pattern of files: there are some three-line files that consist of two blank lines followed by a line that says "Sent from my iPhone" that have slipped into my data. So we make the following change to the program to deal with these files as well.

```
           lines = list()
           for line in fhand:
               lines.append(line)
           if len(lines) == 3 and lines[2].startswith('Sent from my iPhone'):
               continue
           if len(lines) > 1:
```

```
            print len(lines), thefile
            print lines[:4]
```

We simply check if we have a three-line file, and if the third line starts with the specified text, we skip it.

Now when we run the program, we only see four remaining multi-line files and all of those files look pretty reasonable:

```
python txtcheck2.py
3 ./2004/03/22-03-04_2015.txt
['Little horse rider\r\n', '\r\n', '\r']
2 ./2004/11/30-11-04_1834001.txt
['Testing 123.\n', '\n']
2 ./2006/03/17-03-06_1806001.txt
['On the road again...\r\n', '\r\n']
2 ./2006/03/24-03-06_1740001.txt
['On the road again...\r\n', '\r\n']
```

If you look at the overall pattern of this program, we have successively refined how we accept or reject files and once we found a pattern that was "bad" we used `continue` to skip the bad files so we could refine our code to find more file patterns that were bad.

Now we are getting ready to delete the files, so we are going to flip the logic and instead of printing out the remaining good files, we will print out the "bad" files that we are about to delete.

```
import os
from os.path import join
for (dirname, dirs, files) in os.walk('.'):
   for filename in files:
       if filename.endswith('.txt') :
           thefile = os.path.join(dirname,filename)
           size = os.path.getsize(thefile)
           if size == 2578 or size == 2565:
               print 'T-Mobile:',thefile
               continue
           fhand = open(thefile,'r')
           lines = list()
           for line in fhand:
               lines.append(line)
           fhand.close()
           if len(lines) == 3 and lines[2].startswith('Sent from my iPhone'):
               print 'iPhone:', thefile
               continue
```

We can now see a list of candidate files that we are about to delete and why these files are up for deleting. The program produces the following output:

```
python txtcheck3.py
...
T-Mobile: ./2006/05/31-05-06_1540001.txt
T-Mobile: ./2006/05/31-05-06_1648001.txt
iPhone: ./2007/09/15-09-07_074202_03.txt
iPhone: ./2007/09/15-09-07_144641_01.txt
iPhone: ./2007/09/19-09-07_124857_01.txt
...
```

We can spot-check these files to make sure that we did not inadvertently end up introducing a bug in our program or perhaps our logic caught some files we did not want to catch.

Once we are satisfied that this is the list of files we want to delete, we make the following change to the program:

```
           if size == 2578 or size == 2565:
               print 'T-Mobile:',thefile
               os.remove(thefile)
               continue
...
           if len(lines) == 3 and lines[2].startswith('Sent from my iPhone'):
               print 'iPhone:', thefile
               os.remove(thefile)
               continue
```

In this version of the program, we will both print the file out and remove the bad files using `os.remove`.

```
python txtdelete.py
T-Mobile: ./2005/01/02-01-05_1356001.txt
T-Mobile: ./2005/01/02-01-05_1858001.txt
...
```

Just for fun, run the program a second time and it will produce no output since the bad files are already gone.

If we re-run `txtcount.py` we can see that we have removed 899 bad files:

```
python txtcount.py
Files: 1018
```

In this section, we have followed a sequence where we use Python to first look through directories and files seeking patterns. We slowly use Python to help determine what we want to do to clean up our directories. Once we figure out which files are good and which files are not useful, we use Python to delete the files and perform the cleanup.

The problem you may need to solve can either be quite simple and might only depend on looking at the names of files, or perhaps you need to read every single file and look for patterns within the files. Sometimes you will need to read all the files and make a change to some of the files. All of these are pretty straightforward once you understand how `os.walk` and the other `os` utilities can be used.

16.3 Command line arguments

In earlier chapters, we had a number of programs that prompted for a file name using `raw_input` and then read data from the file and processed the data as follows:

```
name = raw_input('Enter file:')
handle = open(name, 'r')
```

```
text = handle.read()
...
```

We can simplify this program a bit by taking the file name from the command line when we start Python. Up to now, we simply run our Python programs and respond to the prompts as as follows:

```
python words.py
Enter file: mbox-short.txt
...
```

We can place additional strings after the Python file and access those **command line arguments** in our Python program. Here is a simple program that demonstrates reading arguments from the command line:

```
import sys
print 'Count:', len(sys.argv)
print 'Type:', type(sys.argv)
for arg in sys.argv:
   print 'Argument:', arg
```

The contents of `sys.argv` are a list of strings where the first string is the name of the Python program and the remaining strings are the arguments on the command line after the Python file.

The following shows our program reading several command line arguments from the command line:

```
python argtest.py hello there
Count: 3
Type: <type 'list'>
Argument: argtest.py
Argument: hello
Argument: there
```

There are three arguments are passed into our program as a three-element list. The first element of the list is the file name (argtest.py) and the others are the two command line arguments after the file name.

We can rewrite our program to read the file, taking the file name from the command line argument as follows:

```
import sys

name = sys.argv[1]
handle = open(name, 'r')
text = handle.read()
print name, 'is', len(text), 'bytes'
```

We take the second command line argument as the name of the file (skipping past the program name in the `[0]` entry). We open the file and read the contents as follows:

```
python argfile.py mbox-short.txt
mbox-short.txt is 94626 bytes
```

Using command line arguments as input can make it easier to reuse your Python programs especially when you only need to input one or two strings.

16.4 Pipes

Most operating systems provide a command-line interface, also known as a **shell**. Shells usually provide commands to navigate the file system and launch applications. For example, in Unix, you can change directories with `cd`, display the contents of a directory with `ls`, and launch a web browser by typing (for example) `firefox`.

Any program that you can launch from the shell can also be launched from Python using a **pipe**. A pipe is an object that represents a running process.

For example, the Unix command[1] `ls -l` normally displays the contents of the current directory (in long format). You can launch `ls` with `os.popen`:

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

The argument is a string that contains a shell command. The return value is a file pointer that behaves just like an open file. You can read the output from the `ls` process one line at a time with `readline` or get the whole thing at once with `read`:

```
>>> res = fp.read()
```

When you are done, you close the pipe like a file:

```
>>> stat = fp.close()
>>> print stat
None
```

The return value is the final status of the `ls` process; `None` means that it ended normally (with no errors).

16.5 Glossary

absolute path:
A string that describes where a file or directory is stored that starts at the "top of the tree of directories" so that it can be used to access the file or directory, regardless of the current working directory.

checksum:
See also **hashing**. The term "checksum" comes from the need to verify if data was garbled as it was sent across a network or written to a backup medium and then read back in. When the data is written or sent, the sending system computes a checksum and also sends the checksum. When the data is read or received, the receiving system re-computes the checksum from the received data and compares it to the received checksum. If the checksums do not match, we must assume that the data was garbled as it was transferred.

command line argument:
Parameters on the command line after the Python file name.

current working directory:
The current directory that you are "in". You can change your working directory using the `cd` command on most systems in their command-line interfaces. When you open a file in Python using just the file

name with no path information the file must be in the current working directory where you are running the program.

hashing:
Reading through a potentially large amount of data and producing a unique checksum for the data. The best hash functions produce very few "collisions" where you can give two different streams of data to the hash function and get back the same hash. MD5, SHA1, and SHA256 are examples of commonly used hash functions.

pipe:
A pipe is a connection to a running program. Using a pipe, you can write a program to send data to another program or receive data from that program. A pipe is similar to a **socket** except that a pipe can only be used to connect programs running on the same computer (i.e. not across a network).

relative path:
A string that describes where a file or directory is stored relative to the current working directory.

shell:
A command-line interface to an operating system. Also called a "terminal program" in some systems. In this interface you type a command and parameters on a line and press "enter" to execute the command.

walk:
A term we use to describe the notion of visiting the entire tree of directories, sub-directories, sub-sub-directories, until we have visited the all of the directories. We call this "walking the directory tree".

16.6 Exercises

Exercise 1

In a large collection of MP3 files there may be more than one copy of the same song, stored in different directories or with different file names. The goal of this exercise is to search for these duplicates.

1. *Write a program that walks a directory and all of its sub-directories for all files with a given suffix (like `.mp3`) and lists pairs of files with that are the same size. Hint: Use a dictionary where the key of the dictionary is the size of the file from `os.path.getsize` and the value in the dictionary is the path name concatenated with the file name. As you encounter each file check to see if you already have a file that has the same size as the current file. If so, you have a duplicate size file and print out the file size and the two files names (one from the hash and the other file you are looking at).*

2. *Adapt the previous program to look for files that have duplicate content using a hashing or **checksum** algorithm. For example, MD5 (Message-Digest algorithm 5) takes an arbitrarily-long "message" and returns a 128-bit "checksum." The probability is very small that two files with different contents will return the same checksum.*

   You can read about MD5 at `wikipedia.org/wiki/Md5`. The following code snippet opens a file, reads it and computes its checksum.

```
import hashlib
...
        fhand = open(thefile,'r')
        data = fhand.read()
        fhand.close()
        checksum = hashlib.md5(data).hexdigest()
```

   You should create a dictionary where the checksum is the key and the file name is the value. When you compute a checksum and it is already in the dictionary as a key, you have two files with duplicate content so print out the file in the dictionary and the file you just read. Here is some sample output from a run in a folder of image files:

```
./2004/11/15-11-04_0923001.jpg ./2004/11/15-11-04_1016001.jpg
./2005/06/28-06-05_1500001.jpg ./2005/06/28-06-05_1502001.jpg
./2006/08/11-08-06_205948_01.jpg ./2006/08/12-08-06_155318_02.jpg
```

   Apparently I sometimes sent the same photo more than once or made a copy of a photo from time to time without deleting the original.

1

When using pipes to talk to operating system commands like `ls`, it is important for you to know which operating system you are using and only open pipes to commands that are supported on your operating system.