정보교육을 위한 파이썬

정보 탐색

Version 0.0.9-d2

저자: Charles Severance
번역: 이광춘, 한정수
(xwmooc)

Printing history:

October 2013: Major revision to Chapters 13 and 14 to switch to JSON and use OAuth. Added new chapter on Visualization.

September 2013: Published book on Amazon CreateSpace

January 2010: Published book using the University of Michigan Espresso Book machine.

December 2009: Major revision to chapters 2-10 from *Think Python: How to Think Like a Computer Scientist* and writing chapters 1 and 11-15 to produce *Python for Informatics: Exploring Information*

June 2008: Major revision, changed title to *Think Python: How to Think Like a Computer Scientist*.

August 2007: Major revision, changed title to *How to Think Like a (Python) Programmer*.

April 2002: First edition of *How to Think Like a Computer Scientist*.

The LaTeX source for the *Think Python: How to Think Like a Computer Scientist* version of this book is available from `http://www.thinkpython.com`.

Chapter 1

컴퓨터의 일반적인 작업 자동화

파일, 네트워크, 서비스, 그리고 데이터베이스에서 데이터를 읽어왔다. 파이썬은 또한 여러분의 컴퓨터의 디렉토리와 폴더를 훑어서 파일도 읽어온다.

이번 장에서, 여러분의 로컬 컴퓨터를 스캔하고 각 파일에 대해서 연산을 수행하는 프로그램을 작성한다. 파일은 디렉토리(또한 "폴더"라고도 부른다.)에 정렬되어 보관된다. 간단한 파이썬 스크립트가 전체 로컬 컴퓨터나 디렉토리를 여기저기 뒤져야되는 수백 수천개 파일에 대한 단순한 작업을 짧게 수행한다.

트리상의 디렉토리나 파일을 여기저기 돌아다니기 위해서 `os.walk`과 `for` 루프를 사용한다. `open`이 파일의 콘텐츠를 읽는 루프를 작성하는 것과 비슷하게, `socket`은 네트워크 연결된 콘텐츠를 읽는 루프를 작성하고, `urllib`는 웹문서를 열어 콘텐츠를 루프를 통해서 읽어오게 한다.

1.1 파일 이름과 경로

모든 실행 프로그램은 "현재 디렉토리(current directory)"가 있고 대부분의 운영체제에 디폴트 디렉토리다. 예를 들어 읽기 위해서 파일을 연다면, 파이썬은 현재 디렉토리에서 파일을 찾는다.

`os` 모듈(os는 "운영체제(operating system)"의 약자)은 파일과 디렉토리를 작업하는 함수를 제공한다. `os.getcwd`은 현재 디렉토리의 이름을 반환한다.

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

`cwd` 는 **current working directory**의 약자로 현재 작업 디렉토리다. 예제의 결과는 `/Users/csev`인데 `csev` 사용자의 홈 디렉토리가 된다.

파일을 식별하는 `cwd` 같은 문자열을 경로(path)라고 부른다. **상대경로(relative path)**는 현재 디렉토리에서 시작하고, **절대경로(absolute path)**는 파일 시스템의 가장 최상단의 디렉토리에서 시작한다.

지금까지 살펴본 경로는 간단한 파일 이름이여서, 현재 디렉토리에서 상대적이다. 파일의 절대 경로를 알아내기 위해서 os.path.abspath을 사용한다.

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

os.path.exists은 파일이나 디렉토리가 존재하는지 검사한다.

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

만약 존재하면, os.path.isdir이 디렉토리인지 검사한다.

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

마찬가지로 os.path.isfile은 파일인지를 검사한다.

os.listdir은 주어진 디렉토리에 파일 리스트(그리고 다른 디렉토리)를 반환한다.

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

1.2 예제: 사진 디렉토리 정리하기

얼마 전에 핸드폰에서 사진을 받아서 서버에 저장하는 플릭커(Flickr)와 유사한 소프트웨어를 개발했다. 플릭커가 존재하기 전에 작성해고, 플릭커가 생긴 이후에도 계속해서 사용했다. 왜냐하면 영원히 원본을 보관하고 싶어서다.

문자메시지로 한줄 텍스트 문자나 전자우편 주소의 제목줄도 보낼 수 있다. 사진 파일과 마찬가지로 텍스프 파일형식의 메시지를 동일한 디렉토리에 저자했다. 사진을 찍은 월, 년, 일, 그리고 시간에 기초한 디렉토리 구조다. 다음은 사진 한장과 설명 텍스트를 가진 예제다.

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

7년이 지난 후에, 정말 많은 사진과 짧은 설명문이 생겼다. 시간이 지남에 따라 핸드폰을 바꿈에 따라, 메시지에서 짧은 설명문을 뽑아내는 코드가 잘 동작하지 않고 짧은 설명문 대신에 서버에 쓸모없는 데이터를 추가했다.

파일을 훑어서 어느 텍스트 파일이 정말 짧은 설명문이고, 어는 것이 쓰레기인지 찾아서 잘못된 파일은 삭제하고 싶었다. 첫번째 할일은 다음 프로그램을 사용하여 폴더에 얼마나 많은 텍스트 파일이 있는지 목록을 얻는 것이다.

```
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
```

이것을 가능하게 하는 가장 중요한 코드는 파이썬 os.walk 라이브러리다. os.walk을 호출하고 시작 디렉토리를 주면, 재귀적으로 모든 디렉토리와 하위 디렉토리를 훑는다. "." 문자열은 현재 디렉토리에서 시작해서 하위 디렉토리로 훑는 것을 표시한다. 매번 디렉토리에 도착하면, for 문 본문의 튜플에서 3개의 값을 얻는다. 첫번째 값은 현재 디렉토리 이름, 두번째 값은 현재 디렉토리의 하위 디렉토리 리스트, 그리고 세번째 값은 현재 디렉토리 파일 리스트다.

명시적으로 하위 디렉토리 각각을 살펴보지 않는다. 왜냐하면 os.walk가 자동으로 모든 폴더를 방문할 것이기 때문이다. 하지만, 각 파일을 살펴보고 싶기 때문에, 간단한 for 루프를 작성해서 현재 디렉토리에 파일 각각을 조사한다. ".txt"로 끝나는 파일이 있는지 확인한다. 접미사 ".txt"로 끝나는 전체 디렉토리 트리를 훑어서 파일의 숫자를 카운트한다.

얼마나 많은 파일이 ".txt" 확장자로 끝나는지 감을 잡았으면, 다음 일은 자동적으로 어느 파일이 정상이고, 어느 파일이 문제가 있는지를 파이썬에서 결정하는 것이다. 간단한 프로그램을 작성해서 파일과 파일의 크기를 출력한다.

```
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
```

파일을 단순히 카운트하는 대신에 os.path.join을 사용하여 디렉토리 안에서 디렉토리 이름과 파일 이름을 합쳐서 파일 이름을 생성한다. 문자열 합치기 대신에 os.path.join을 사용하는 것이 중요한데 왜냐하면 파일 경로를 나타내기 위해서 윈도우에서는 파일 경로를 생성하기 위해서 역슬래쉬(\)를 사용하고, 리눅스나 애플에서는 슬래쉬 (/)를 사용하기 때문이다. os.path.join은 이러한 차이를 알고 어느 운영체제에서 동작하는지 알고 시스템에 따라 적절한 합치기 작업을 수행한다. 그래서 동일한 파이썬 코드가 윈도우나 유닉스계열 시스템에도 실행된다.

디렉토리 경로를 가진 전체 파일 이름을 갖게 되면, os.path.getsize 유틸리티를 사용해서 크기를 얻고 출력해서 다음 결과값을 만들어 낸다.

```
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
...
```

출력값을 스캔하면, 몇몇 파일은 매우 짧고, 다른 많은 파일은 매우 큰데 동일한 크기(2578, 2565)임을 볼 수 있다. 수작업으로 몇개의 큰 파일을 살펴보면, T-Mobile 핸드폰에서 보내지는 전자우편에 함께 오는 일반적인 동일한 HTML을 가진 것임을 알 수 있다.

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

파일을 대충 살펴보면, 파일에 그다지 유용한 정보가 없으므로, 삭제한다.

하지만, 파일을 삭제하기 전에, 한 줄이 이상인 파일을 찾고 내용을 출력하는 프로그램을 작성한다. 2578 혹은 2565 문자길이를 가진 파일을 보여주지는 않는다. 왜냐하면, 파일에 더 이상 유용한 정보가 없음을 알기 때문이다.

그래서 다음과 같이 프로그램을 작성한다.

```
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]
```

`continue`를 사용하여 두 "잘못된 크기" 파일을 건너뛰고, 나머지 파일을 열고, 파이썬 리스트에 파일 라인을 읽는다. 만약 파일이 하나 이상의 라인이면, 파일에 얼마나 많은 라인이 있는지 출력하고, 첫 세줄을 출력한다.

두개의 잘못된 파일 크기를 제외하고, 모든 한줄짜리 파일이 정상적이라고 가정하고나면 깨끗하게 정리된 파일을 생성된다.

```
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
...
```

하지만, 한가지 더 성가신 패턴의 파일이 있다. 두 공백 라인과 “Sent from my iPhone”으로 구성된 3줄짜리 파일이다. 프로그램을 다음과 같이 변경하여 이러한 파일도 처리하게 한다.

```
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]
```

단순하게 3줄짜리 파일이 있는지 검사하고 만약 지정된 텍스트로 세번째 라인이 시작한다면, 건너뛴다.

이제 프로그램을 실행하면, 단지 4개의 다중 라인 파일만을 보게되고 모든 파일이 잘 처리된 것으로 보인다.

```
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']
```

프로그램의 전반적인 패턴을 살펴보면, 연속적으로 파일을 어떻게 승인할지와 거절할지를 정교화했고, ”잘못된” 패턴을 발견하면 `continue`를 사용해서 잘못된 파일을 건너뛰게 했다. 그래서 잘못된 더 많은 파일 패턴을 발견하도록 코드를 정교화했다.

이제 파일을 삭제할 준비가 되었다. 로직을 바꿔서, 나머지 올바른 파일을 출력하는 대신에, 삭제할 ”잘못된” 파일만을 출력한다.

```
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
```

이제 삭제할 대상 목록과 왜 이 파일이 삭제 대상으로 나왔는지를 볼 수 있다. 프로그램은 다음 출력을 생성한다.

```
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
...
```

무작위로 파일을 검사해서 프로그램에 버그가 우연하게 들어가 있는지 혹은 원치 않는 파일이 작성한 프로그램 로직에 끌려들어 갔는지 확인할 수 있다.

결과값에 만족하고, 다음이 삭제할 파일 목록임으로, 프로그램에 다음과 같은 변경을 한다.

```
        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
```

이번 버젼의 프로그램에서 파일을 출력하고 `os.remove`을 사용하여 잘못된 파일을 삭제한다.

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

재미로, 프로그램을 두번 실행하게 되면 모든 잘못된 파일이 삭제되어서 출력값이 없다.

`txtcount.py`을 다시 실행하면, 899 잘못된 파일이 삭제되었음을 알 수 있다.

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

이번 장에서 일련의 절차를 따라서 파이썬을 사용하여 디렉토리와 파일을 검색하는 패턴을 살펴봤다. 천천히 파이썬을 사용해서 디렉토리를 정리하기 위해서 무엇을 할지를 결정했다. 어느 파일이 좋고 어느 파일이 유용하지 않는지 파악한 후에 파이썬을 사용해서 파일을 삭제하고 파일 정리를 수행했다.

해결하려고 하는 문제가 무척 간단하여 파일의 이름만을 살펴보는 것에 달려 있을 수 있다. 혹은 모든 파일을 읽고 파일 내부에 패턴을 찾을 필요가 있다. 때때로, 모든 파일을 일고, 파일의 일부에 변경이 필요할지도 모른다. `os.walk`와 다른 `os` 유틸리티가 어떻게 사용되는지 이해하기만 하면 이 모든 것은 매우 명확한다.

1.3 명령 줄 인수

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 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.

1.4 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()
```

[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.

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).

1.5 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".

1.6 Exercises

Exercise 1.1 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.