Files

Sutee Sudprasert

เนื้อหาส่วนใหญ่ ลอกมาจาก Mark Pilgrim. Dive into Python
http://greenteapress.com/thinkpython/html/book015.html

Files

- Persistent
    - the programs keep at least some of their data in permanent storage and if they shut down and restart, they pick up where they left off.
    - One of the simplest ways for programs to maintain their data is by reading and writing text files.

Reading

- Open

```
>>> fin = open('words.txt')
>>> print fin
<open file 'words.txt', mode 'r' at 0xb7f4b380>
```

- Read

```
>>> fin.readline()
'aa\r\n'
```

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print word
```

Writing

- Open

```
>>> fout = open('output.txt', 'w')
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

- Write

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
```

- Close

```
>>> fout.close()
```

Open

- Append

```
>>> fout = open('output.txt', 'w+')
>>> print fout
<open file 'output.txt', mode 'w+' at 0x38f180>
```

- Binary

```
>>> fout = open('output.txt', 'wb')
>>> print fout
<open file 'output.txt', mode 'wb' at 0x38f180>
```

Using `with` statement

- Python 2.5+

```
with open(“file.txt”) as fin:
    for line in fin:
        word = line.strip()
        print word
```

- `fin` will have been automatically closed, even if the for loop raised an exception partway through the block.

Format operator

- % is the modulus operator when applied to integer, but when the first operand is a string, % is the format operator

```
>>> camels = 42
>>> '%d' % camels
'42'
```

  - the first operand is the **format string**, which contains one or more **format sequences**
  - the second operand is formatted following the format string
  - the result is a string

Format operator

- '%d' to format an integer, '%g' to format a floating-point number , and '%s' to format a string

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

- The number of elements in the tuple has to match the number of format sequences in the string.

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

http://docs.python.org/release/2.5.2/lib/typesseq-strings.html (more details about format string)

Filenames and paths

- The `os` module provides functions for working with files and directories

```
>>> import os
>>> cwd = os.getcwd()
>>> print cwd
/home/dinsdale
```

returns the name of the current directory

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

returns the absolute path to a file

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

checks whether a file or directory exists

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

checks whether it’s a directory

Filenames and paths

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

returns a list of the files (and other directories) in the given directory

```
def walk(dir):
    for name in os.listdir(dir):
        path = os.path.join(dir, name)

        if os.path.isfile(path):
            print path
        else:
            walk(path)
```

The `os` module provides a function called `walk` that is similar to this one but more versatile.

the following example "walks" through a directory, prints the names of all the files, and calls itself recursively on all the directories

Filenames and paths

```
>>> os.path.join(‘images’,‘dogs’,‘test.png’)
‘images/dogs/test.png’ # Mac and Linux
‘images\dogs\test.png’ # Windows
```

Dictionaries and files manipulation

```
mkdir(path[, mode])
link(src, dst)
symlink(src, dst)
remove(path)
rename(src, dst)
rmdir(path)
```

Databases (a simple one)

- Most databases are organized like a dictionary in the sense that they map from keys to values.
  - The biggest difference is that the database is on disk (or other permanent storage), so it persists after the program ends.
- The module `anydbm` provides an interface for creating and updating database files

Databases

- Opening a database is similar to opening other files:

```
>>> import anydbm
>>> db = anydbm.open('captions.db', 'c')
```

'c' means that the database should be created if it doesn't already exist.

- If you create a new item, `anydbm` updates the database file.

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

- When you access one of the items, anydbm reads the file:

```
>>> print db['cleese.png']
Photo of John Cleese.
```

Databases

- If you make another assignment to an existing key, anydbm replaces the old value:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> print db['cleese.png']
Photo of John Cleese doing a silly walk.
```

- Many dictionary methods, like keys and items, also work with database objects. So does iteration with a for statement.

```
for key in db:
    print key
```

- you should close the database when you are done:

```
>>> db.close()
```

Pickling

- A limitation of anydbm is that the keys and values have to be strings. If you try to use any other type, you get an error.
- The pickle module can translate **almost** any type of object into a string suitable for storage, and then translates strings back into objects.

Pickling

- `pickle.dumps` takes an object as a parameter and returns a string representation (dumps is short for “dump string”):

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
'(lp0\nI1\naI2\naI3\na.'
```

- `pickle.dump` takes an object and a file object as parameters and return `True` if it succeeds otherwise `False`.

```
>>> t = [1, 2, 3]
>>> with open(‘list.pkl’,’w’) as fout:
>>>     pickle.dump(t, fout)
```

Pickling

- `pickle.loads` ("load string") reconstitutes the object:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> print t2
[1, 2, 3]
```

- `pickle.load` reconstitutes the object from an opened file object:

```
>>> with open('list.pkl') as fin:
>>>   t2 = pickle.load(fin)
>>>   print t2
[1, 2, 3]
```

Pipes

- 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.
- If you don't want to read the output or pass any parameter to the process, you can use:

```
>>> import subprocess
>>> subprocess.call([‘ls’,’-l’])
total 8
-rw-r--r--  1 sutee  staff     0 Jul 19 15:19 output.txt
-rw-r--r--@ 1 sutee  staff  2542 Jul 19 13:25 thainum.py
0
```

Pipes

- Read from `stdout`

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
```

- Read from `stdout` and write to `stdin`

```
>>> from subprocess import Popen, PIPE
>>> p = Popen([“wc”], stdin=PIPE, stdout=PIPE)
>>> p.stdin.write(“test test\ntest test”)
>>> p.stdin.close()
>>> res = p.stdout.read()
>>> stat = p.stdout.close()
```

Writing modules

- Any file that contains Python code can be imported as a module.
- For example: wc.py

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count
print linecount('wc.py')
```

- You can import it like this:

```
>>> import wc
7
>>> wc.linecount('wc.py')
7
```

Writing modules

- The only problem with this example is that when you import the module it executes the test code at the bottom.
- Normally when you import a module, it defines new functions but it doesn't execute them.
- Programs that will be imported as modules often use the following idiom:

```
if __name__ == '__main__':
    print linecount('wc.py')
```