Exercise 7 – Data Storage and File Handling

Objective

To use some of the Python 3 file handling methods, as well as the pickle and gzip modules.

Questions

1. Write a Python script to list all the unused port numbers in the /etc/services file between 1 and 200

   Steps:

   Become familiar with the input file - view it first

   Write the main code to read the services file one line at a time

   Use string functions or a regular expression to:

   Ignore lines starting with a # comment character

   Ignore lines that just consist of "white-space"

   The /etc/services has several columns separated by white-space

   - Use split or a regular expression to isolate the port/protocol field
   - Use another split or regular expression to isolate the port number
   - Don't forget to stop at port number 200!
   - Note that many port numbers have > 1 entry

   On Windows the file is in 'C:\WINDOWS\system32\drivers\etc\services' or in 'C:\WINNT\system32\drivers\etc\services'.

   On OSX the file has unused ports marked as 'Unassigned'. Therefore, we have an addition requirement: ignore all lines that start with the comment delimiter '#'.

   Many port numbers have more than one entry in the file, but you may assume they are in order.

   Hints: Open the file.
   Read the file line-by-line using a for loop.
   Consider using a set or a dictionary to hold the port numbers.

Be careful of comparing strings and int - you will have to convert the port number to an integer.

2. Using the data in **country.txt**, construct a Python dictionary where the country name is the key and the other record details are stored in a list as the value. Store (pickle) this dictionary into a file named 'country.p'.

   Notice the size of the file compared to the original, and then change the program to use gzip.

3. Now write a program which reads the pickled dictionary and displays it onto the console.

   If time allows, convert your pickle to use a shelve.

If time allows...

4. This exercise uses **messier.txt**, which was used in a previous optional exercise (you do not need to have completed that exercise to do this one).

   This file contains details of Messier celestial objects that are identified by a Messier number, the first field in the file.

   The aim is to access the records in the file randomly, using seek(). First construct an index (could be in a list or a dictionary) which consists of the file position (use tell()) of each record. The key is the first field, the Messier number, which is prefixed M (ignore any lines that do not start with 'M').

   Now prompt the use to enter a Messier number, with or without the 'M', and display the record for that celestial object.

   The file **messier.txt** has a character which uses **'latin_1'**, how do you cope with that?

5. You may recall an exercise from Chapter 5 Collections that timed various ways of searching for the word 'Zulu', using the program **Ex5_4.py**. The fastest technique by far was to use a dictionary.

   Modify **Ex5_4.py** to use a shelve, preferably by copying the code from your dictionary implementation and modifying the copy. If you did not complete that exercise, then use the solution code in solutions/05 Collections.

You'll find that the shelve is considerably slower than other techniques. However, place an additional start_timer()/end_timer() around the shelve creation (including loading the data into the dictionary). This should give two sets of times for using a shelve, loading and searching (which is repeated in a loop).

Where is the biggest overhead? Is this a reasonable test?

Solutions

Question 1

This solution uses regular expressions and sets. A common mistake with this approach is to forget to convert the captured port number to an int, required since range returns an integer.

```
import sys
import re

if sys.platform == 'win32':
   file = r'C:\WINDOWS\system32\drivers\etc\services'
else:
   file = '/etc/services'

ports = set()

for line in open(file, 'r'):
   m = re.search(r'(\d+)/(udp|tcp)', line)
   if m:
      port = int(m.group(1)) # Or m.groups()[0])
      if port > 200: break
      ports.add(port)

# Subtract used port numbers from full set of ports
print(set(range(1, 201)) - ports)
```

Questions 2 & 3

```
import pickle
import gzip
import shelve

# Using a compressed pickle.
country_dict = {}
for line in open('country.txt', 'r'):
    name, *row = line.split(',')
    country_dict[name] = row

outp = gzip.open('country.p', 'wb')
pickle.dump(country_dict, outp)
outp.close()

# Using a shelve.
db = shelve.open('country')
for country in country_dict.keys():
    db[country] = country_dict[country]

db.close()
db = shelve.open('country')
print(db['Belgium'])
db.close()
```

If time allows...

Question 4

```
# Construct an index.
index = []
fh_in = open('messier.txt', 'r', encoding='latin_1')

while True:
   line = fh_in.readline()
   if not line: break
   if line.startswith('M'):
            num = line[1:6].rstrip()
      index.append(fh_in.tell() - len(line))

while True:
   num = input('Enter a Messier number to exit): ')
   if num.startswith('M'):
      num = int(num[1:])
   elif num:
      num = int(num)
   else:
      num = 0

   if num < 1: break
   num -= 1

   fh_in.seek(index[num])
   print(fh_in.readline())
```

Question 5

```
import shelve

def shelve_func():
   global shelve_dict
   return shelve_dict['Zulu'] + 1


i = 0
start_timer()
shelve_dict = shelve.open('shelve_dict')

for key in words:
   shelve_dict[key] = i
   i += 1
```

```
end_timer('Shelve creation')


start_timer()

for i in range(0, LOOP_COUNT):
    line = shelve_func();

shelve_dict.close()
end_timer('Shelved dictionary')

print('Dictionary line number: ', line)
```

The timings obtained (on a test machine) were:

```
Brute_force : 1.014 seconds
Brute_force line number: 45400
Index       : 0.312 seconds
Index line number: 45400
In          : 0.296 seconds
Dictionary  : 0.016 seconds
Dictionary line number: 45400
Shelve creation: 38.517 seconds
Shelved dictionary: 0.468 seconds
Dictionary line number: 45400
```

The timings are not surprising, although Python 3.2 performance does not compare well with Python 2.7, which only takes round 1.7 seconds to create the same database. The Python 2 database file sizes are considerably smaller than those used by Python 3.
After creation, caching means that we are essentially dealing with an in-memory dictionary, nevertheless, it is still slower than a conventional dictionary. The dataset is probably too small to meaningfully measure retrieval overheads. Since record sizes, and therefore the number of I/O transfers, will vary considerably between applications, a general measure with such a simple structure is not a reliable guide.