Buğra Gedik's Courses @ Bilkent University

cs315:fall2016:hw1

In this part, you will learn how to use the scripting language called Python. In particular, we will be using Python 2.7 series (there is also Python 3 series, but that is not yet well adopted in the industry).

Learn about Python here [http://docs.python.org/2/tutorial/].

Python has a library called NumPy, which provides support for multi-dimentional arrays and matrices.

Learn about NumPy. There are various tutorials out there (one [http://cs231n.github.io/python-numpy-tutorial/], another [http://www.cs.man.ac.uk/~barry/mydocs/MyCOMP28512/MS15_Notes/PyRefs/Tentative_NumPy_Tutorial.pdf]).

This homework has 5 parts.

Part A) Arrays and copy semantics

Consider the following Python code segment, which uses built-in Python lists and NumPy lists to perform similar operations, albeit with differing results.

Using built-in Python lists:

```
>>> data = [1, 2, 3, 4]
>>> print data
[1, 2, 3, 4]
>>> otherData = data
>>> otherData[1] = -2
>>> print otherData
[1, -2, 3, 4]
>>> print data
[1, -2, 3, 4]
>>>
>>> otherData = data[1:3]
>>> print otherData
[-2, 3]
>>> otherData[0] = 0
>>> print otherData
[0, 3]
>>> print data
[1, -2, 3, 4]
```

Using NumPy arrays:

```
>>> import numpy as np
>>> data = np.array([1, 2, 3, 4])
>>> print data
[1 2 3 4]
>>> otherData = data
>>> otherData[1] = -2
>>> print otherData
[1 -2  3  4]
>>> print data
[1 -2  3  4]
>>>
```

```
>>> otherData = data[1:3]
>>> print otherData
[-2  3]
>>> otherData[0] = 0
>>> print otherData
[0 3]
>>> print data
[1 0 3 4]
```

Describe similarities and differences between copying and assignment semantics of built-in Python lists and NumPy arrays. Explain why the code behaves differently for the two.

Part B) Matrices

NumPy also supports matrices. However, there are some important differences between two-dimentional arrays and matrices. Consider the following two code segments that are similar, but produce different results:

Using NumPy two-dimentional arrays:

```
>>> A = np.array([[1,2], [3,4]])
>>> B = np.array([[2,1], [-1,2]])
>>> A * B
array([[ 2,  2],
       [-3,  8]])
>>> A ** 3
array([[ 1,  8],
       [27, 64]])
```

Using NumPy matrices:

```
>>> A = np.matrix([[1,2], [3,4]])
>>> B = np.matrix([[2,1], [-1,2]])
>>> A * B
matrix([[ 0,  5],
        [ 2, 11]])
>>> A**3
matrix([[ 37,  54],
        [ 81, 118]])
```

Describe similarities and differences between NumPy two-dimentional arrays and matrices. Explain why the code behaves differently for the two.

Part C) Reading from files

Assume you have a file that contains an undirected graph. The graph is described according to the following BNF grammar:

```
<graph> -> <graph_header> <graph_body>
<graph_header> -> "[num_nodes]" NUMBER
<graph_body> -> "[edges]" <edges>
<edges> -> <edges> <edge> | <edge>
<edge> -> NUMBER -- NUMBER
```

There is only a single token, which is defined as:

```
NUMBER: [0-9]+
```

Here is an example input:

```
[num_nodes]
5
```

```
[edges]
0 -- 3
0 -- 4
1 -- 2
1 -- 3
2 -- 4
3 -- 3
```

You can assume that the vertex indices start from 0. A graph like this can be represented as a matrix, as follows:

```
0 0 0 1 1
0 0 1 1 0
0 1 0 0 1
1 1 0 1 0
1 0 1 0 0
```

Note that the matrix is symmetric. We do not need to specify an edge twice, $u \leftrightarrow v$ and $v \leftrightarrow u$, as one implies the other.

Implement the following two functions:

- Given the name of a file that contains a graph, read the file and convert it into an adjacency matrix (using NumPy matrices).
- Given the adjacency matrix representation and a number `m`, for each pair of vertices `(u, v)`, find the number of paths from `u` to `v` with length `m`. Hint: This requires a simple matrix manipulation.

For the first part, you need to study working with files in Python. It is described here [http://docs.python.org/2/tutorial/inputoutput.html#reading-and-writing-files].

Part D) Going large-scale

What if your graph is large, say 10000 vertices. The matrix representation will end up eating a lot of space. This seems somewhat unnecessary, as most of the entries in the matrix will be 0. How can you make such matrices take less space in memory? Is there a Python library that you can locate which can handle this case? Do some research and report your findings.

Part E) Data structures warm-up

Consider the following code that builds up a million accounts with unique ids, random names, and random balances. It then continuously asks the user for a name, and lists the accounts with matching names. Note that more than one account can have the same name.

```
1  #!/usr/bin/env python
2  import random
3  import time
4
5  class Account:
6      def __init__(self, id, name, balance):
7          self.id = id
8          self.name = name
9          self.balance = balance
10
11     def __str__(self):
12         return "[id:%d, name: %s, balance: %s]" % (self.id, self.na
13
14     def getId(self):
15         return self.id
16
17     def getName(self):
18         return self.name
19
```

```
    def getBalance(self):
        return self.balance

    def withdraw(self, amount):
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance

def generateRandomAccounts(n):
    accounts = []
    nameLen = 5
    balanceMax = 1000
    alpha = 'abcdefghijklmnopqrstuvwxyz'
    for id in xrange(1, n+1):
        name = ""
        for i in xrange(0, nameLen):
            name += random.choice(alpha)
        if id==1:
            print "First account name is '%s'" % name
        balance = random.choice(xrange(0, balanceMax))
        account = Account(id, name, balance)
        accounts.append(account)
    return accounts

def findAccountIndices(name, accounts):
    indices = []
    for (index, account) in enumerate(accounts):
        if account.getName() == name:
            indices.append(index)
    return indices

start = time.time()
accounts = generateRandomAccounts(1000000)
end = time.time()
print "Accounts created in %f seconds" % (end-start)
while True:
    name = raw_input("Account name: ")
    if name == "":
        print "Exiting..."
        break
    start = time.time()
    indices = findAccountIndices(name, accounts)
    end = time.time()
    if len(indices)>0:
        for index in indices:
            print "> Account found: %s" % accounts[index]
    else:
        print "> No accounts found."
    print "Search took %f seconds" % (end-start)
```

The code stores all the accounts in a list. When a request is made to search for an account, this list is traversed and matching indices are found. This is performed by the `findAccountIndices` method. Here is a sample run:

```
First account name is 'flkku'
Accounts created in 8.334120 seconds
Account name: dfdfd
> No accounts found.
```

```
Search took 0.280890 seconds
Account name: flkku
> Account found: [id:1, name: flkku, balance: 174]
Search took 0.278653 seconds
```

It looks like our search process is rather slow, as we can only perform around 3.5 queries per second. This is rather slow for a computer and could be a major performance problem for a multi-user system with high throughput of queries. We ask you to use Python dictionaries to speed up this process. In particular, write a `createNameMap` function which will create a mapping from account names to lists of account indices. The modified driver program should look like this:

```
start = time.time()
accounts = generateRandomAccounts(1000000)
end = time.time()
print "Accounts created in %f seconds" % (end-start)
start = time.time()
nameMap = createNameMap(accounts) ##### Build a map to speed things
end = time.time()
print "Name map created in %f seconds" % (end-start)
while True:
    name = raw_input("Account name: ")
    if name == "":
        print "Exiting..."
        break
    start = time.time()
    indices = nameMap.get(name, []) ##### Fast dictionary lookup
    end = time.time()
    if len(indices)>0:
        for index in indices:
            print "> Account found: %s" % accounts[index]
    else:
        print "> No accounts found."
    print "Search took %f seconds" % (end-start)
```

A sample run from the new version is as follows:

```
First account name is 'khdyc'
Accounts created in 8.292552 seconds
Name map created in 1.741065 seconds
Account name: dfdgdg
> No accounts found.
Search took 0.000009 seconds
Account name: khdyc
> Account found: [id:1, name: khdyc, balance: 333]
Search took 0.000012 seconds
```

As you can see, we can now perform around 100,000 searches per second at the cost of some start-up processing (creation of the name map) as well as additional memory used (to keep the name map).

Write the code for the `createNameMap` function.

Logistics

It is best if you install Python and NumPy on your own computer. But you could also find them pre-installed at dijkstra.ug.bcc.bilkent.edu.tr

Put your code and report under a directory named lastname_name_hw1 and make an archive from that directory. Your report should be named lastname_name_hw1.pdf (or .txt). For example, the following Unix commands could be used:

```
mkdir lastname_name_hw1
cd lastname_name_hw1
    ...
    (edit and test your files in this directory)
    ...
cd ..
tar -cvzf lastname_name_hw1.tar.gz lastname_name_hw1
```

Then upload the generated file (named lastname_name_hw1.tar.gz) to Moodle.

Reports in formats other than .txt and .pdf are not accepted.

cs315/fall2016/hw1.txt · Last modified: 2016/10/19 21:27 by bgedik