600-152 Informatics 2 Lecture Notes.

Indexing.

Bernie Pope. 22/3/09.

From data to information

Data becomes information when we do something with it. For instance we might process the data somehow and use the results to answer a question. The question might be something simple, such as

> In which movies has Audrey Hepburn been a member of the cast?

Or the question might be quite complicated, such as

> Is their any correlation between the use of mobile phones and cancer?

Generally speaking, the more (accurate) data we have, the better answers we can derive. We therefore collect lots of data in the hope that it will yield better quality information. Unfortunately this leads is to a conundrum: we can get better information with more data, but more data also takes longer to process because more work has to be done. If it takes too long to answer a question then we may eventually give up, and that would defeat the purpose of collecting the data in the first place! We need some way to balance the tension between the desire for more data and the desire for getting answers in reasonable amounts of time. This is where the study of algorithms and data-structures becomes important. Better algorithms and data-structures allow us to use the facilities of computers more efficiently, which means we can get more done in less time (or perhaps with less memory).

Searching is a fundamental component of data processing, therefore people have spent a lot of time trying to find efficient ways to do it. This lecture concerns the practice of *indexing* as a means to improve the efficiency of search. To demonstrate the key ideas, we will develop a Python program for searching for definitions in a dictionary of computer terms. This example was chosen because it is simple enough to explain in a lecture, but the same ideas can be applied to more sophisticated applications.

In this lecture we will rely heavily on Python's built-in dictionary data-structure which provides a fast mapping between keys and values. We assume that you know how to use Python's dictionaries from your studies in Informatics 1, therefore some revision of the topic may be in order. In the following lectures we will consider how Python's dictionaries might be implemented. One slightly awkward aspect of this lecture is that we are using the word *dictionary* in two different senses:

1. A *thing* which defines the meanings of terms, such as a dictionary book, or a dictionary computer application.
2. The name of the Python data structure which maps keys to values.

It will be helpful to give a name to the application we are creating, so we will henceforth refer to it as 'SearchMe'. Hopefully the remaining uses of the word *dictionary* will be clear from their context.

One of the goals of this lecture is to prepare you for the second and third stages of the project. A number of exercises are posed along the way. They are not mandatory, but they are designed to help you consolidate the material, and should be quite useful to your study of the subject. As always, you can get help from the lecturers if you need it.

The data source

The *Computer Dictionary Project* is a free online dictionary of computer terms:

http://computerdictionary.tsf.org.za/

The dictionary contains over twenty thousand entries, and it can be browsed online at the URL above. The data can also be downloaded as a single large HTML file, and we will use that as the basis of SearchMe. HTML is designed to be a markup language for presentation, but it is cumbersome to search. Therefore, to simplify our task, I have converted it into CSV format, and stored it in a file called `dict.csv`.The full HTML file is about six megabytes in size, whereas `dict.csv` is about half of that, so we get a useful space saving at the same time.

Exercise 1

Download the HTML source of the dictionary from:

http://computerdictionary.tsf.org.za/dictionary/terms/computerdictionary-all.html

Write a Python program to convert the file into CSV format. You may find the Beautiful Soup and csv libraries useful.

Below is a sample of three (non-consecutive) rows from `dict.csv`. Each row has two columns. The first column contains the term being defined, and the second column contains the definition.

$HOME environment variable	The home directory of the current user; the default argument for the cd builtin command.

CSV	Comma Separated Values
yydecode	| decode yEnc archives yydecode works almost identically to the infamous uudecode program, but for yEnc encoded archives. |

As you can see, the terms can contain multiple words and non-alphabetic characters. The definitions are just free-form text with no line breaks. The vertical bar character '|' acts as a kind of quotation mark to prevent commas in the term or definition from being treated as separators.

Exercise 2

In the example table above, why is the definition of yydecode quoted, when the definitions of the other terms are not quoted?

The `csv` library makes it easy to read the contents of CSV files in Python, as the following program demonstrates:

```
import csv

filename = 'dict.csv'
file = open(filename)
reader = csv.reader(file, delimiter=',', quotechar='|')

for row in reader:
    if len(row) >= 2:
        print 'term = %s' % row[0]
        print 'definition = %s' % row[1]

file.close()
```

A simple user interface

The basic structure of the SearchMe program is this (note this code is not complete). Let us suppose this code is stored in a file called `searchme.py`. As the lecture progresses we will make modifications to this code, refining the implementation as we go.

```
def normalise(str):
    return str.strip().lower()

def get_input():
    prompt = 'Enter a query term: '
    input = raw_input(prompt)
```

```
    return normalise(input)

# the search function is not complete
def search(query):
    return None

query = get_input()

while len(query) > 0:
   result = search(query)
   if result == None:
      print 'Search failed'
   else:
      print result
   query = get_input()
```

The main work of the program is performed inside the `while` loop. Each time around the loop it prompts the user to input a query term. It searches for the definition of the query term and displays the result. The loop terminates when the query term is empty. At the moment the `search()` function is not completely defined; we will get to that shortly. The `get_input()` function reads input from the user. It uses the `raw_input()` library function to ask the user for input, and to read their response. The `normalise()` function strips off any leading and trailing white space, and converts the remaining characters to lower case.

Exercise 3

The `normalise()` function transforms the user's input text in a couple of ways:

1. Leading and trailing white space characters are removed.
2. The remaining characters are converted to lower case.

Why is it useful to do these things in the context of SearchMe?

Linear search

Let is now try to implement the `search()` function. It is generally a good programming strategy to start with the simplest and most obvious algorithm that you can think of. Then, once that is working correctly, you can try something more ambitious. As the saying goes: don't bite off more than you can chew!

Linear search embodies the idea of inspecting a sequence of things one after the other, as if they were in a line. In the case of SearchMe, a linear search for a query entails looking at each row of the CSV file until either the query is found in the first column of a row, or all the rows have been inspected.

The code below illustrates our first attempt at defining the `search()` function. To try it out you must paste it into the `searchme.py` program and delete the old, incomplete, definition of `search()`.

```
import csv

def search(query):
    file = open('dict.csv')
    reader = csv.reader(file, delimiter=',', quotechar='|')
    for row in reader:
        if len(row) >= 2 and normalise(row[0]) == query:
            file.close()
            return row[1]
    file.close()
    return None
```

Here is an example execution of the program, input from the user is indicated in **bold** font:

```
Enter a query term: yydecode
decode yEnc archives yydecode works almost identically to the
infamous uudecode program, but for yEnc encoded archives.
Enter a query term: CSV
Comma Separated Values
Enter a query term: Bernie
Search failed
Enter a query term:
```

On the last line the user entered an empty input (by hitting the return key immediately), thus terminating the program.

Exercise 4

In the definition of `search()` above (the one that uses linear search), you can see that it opens the file `dict.csv` and creates a new reader:

```
file = open('dict.csv')
reader = csv.reader(file, delimiter=',', quotechar='|')
```

This seems like it might be inefficient to do this every time the function is called. Perhaps this only needs to be done once? Wouldn't it be better to do it the following way?

```
file = open('dict.csv')
reader = csv.reader(file, delimiter=',', quotechar='|')

def search(query):
    for row in reader:
        if len(row) >= 2 and normalise(row[0]) == query:
            return row[1]
    return None
```

What happens if do it this way? Does it always work properly?

As you can see, linear search gives correct results, and on a small sized input it is fast enough, but it is not an ideal algorithm. Suppose there are *N* rows in our CSV file (for some number *N*). Imagine that we perform a large number of queries using linear search, each time choosing a query term at random from the set of terms defined in the CSV file. That is, we choose query terms which are guaranteed to be found in the file, and we choose them randomly. Suppose also that we time how long each individual call to `search()` takes. We will find that, on average, the time taken is proportional to N/2. The reason is that the average position of a row in the CSV file is exactly in the middle. To find the middle row using linear search we must step over the first *N/2* rows.

Can we do any better than linear search?

Indexing

We can do better than linear search if we construct an (efficient) index of the data. An index is simply a mapping from some set of keys to some set of values. In our case the keys are the terms being defined, and the values are the corresponding definitions of the terms. The purpose of an index is to help us find something in our data without having to look at all of the other entries. Python's dictionaries are the obvious way to represent such an index, and indeed that is exactly what dictionaries are designed to do. The good news for us is that the Python developers have spent a lot of time tuning their implementation of dictionaries to run fast; we enjoy the fruits of their labour.

Below is a function called `make_index()` which constructs an index of the data in the CSV file. The index is represented using a Python dictionary which has terms as its keys and definitions as its values.

```
def make_index():
    index = {}
    file = open('dict.csv')
    reader = csv.reader(file, delimiter=',', quotechar='|')
    for row in reader:
        if len(row) >= 2:
            key = normalise(row[0])
            val = row[1]
```

```
            index[key] = val
    file.close()
    return index
```

In order to use the index we have to make a few minor adjustments to the code in `searchme.py`. The necessary adjustments are shown below. The parts that have changed are indicated by highlighting. Note: the `normalise()` and `get_input()` functions do not change, so we don't show them below. Obviously we would also need to include the definition of the `make_index()` function in the program too.

```
index = make_index()

def search(index,query):
    return index.get(normalise(query))

query = get_input()
while len(query) > 0:
   result = search(index,query)
   if result == None:
      print 'Search failed'
   else:
      print result
   query = get_input()
```

Note that we are careful to only build the index once, rather than every time `search()` is called.

Exercise 5

The new version of the `search()` function above is now nice and simple. The body of the function contains just this line of code:

```
  return index.get(normalise(query))
```

If relies on the `get()` method of dictionaries. What does the `get()` method do? What does it return if its argument is not found in the dictionary?

Obviously it takes time to build the index, and you might wonder whether it is worth the effort. If we only do a small number of searches then it might take more time to build the index than it takes to do linear search. Therefore we only really benefit from indexing if we can use the same index for a large number of searches. Unfortunately this indicates a slight problem in the design of our program. Every time we run `searchme.py` we re-build the index from scratch. Wouldn't it be nice if we could build the index once and then somehow

save it to a file, so we don't have to keep re-building it? Perhaps we should split SearchMe into two parts:

1. Index the data from the CSV file and save the index to a file (do this only once).
2. Perform the interactive search, using the index created by the first part.

How can we save the index to file efficiently?

Pickling

Fortunately Python provides a convenient and efficient way to save the index to a file without having to re-build it every time. This functionality is provided by the `pickle` library (or alternatively the `cPickle` library), via the functions `dump()` and `load()`. The `dump()` function takes two arguments: a Python value and a file. It writes a binary representation of the value into the file. The binary representation is designed to be compact and fast to manipulate. The `load()` function is essentially the inverse, allowing you to retrieve a pickled value from a file.

Exercise 6

Alternatively, we could save the index to file using an XML format. For instance, you might represent the index as a sequence of `<record>` elements, where each record has a `<key>` and `<value>` child.

Design an XML format to do this, and write a Python program to create such a file from the index.

What advantages and disadvantages might this approach have over pickling?

Below is the code for a Python program which creates an index of the CSV file and pickles it to a file for later use. Let us call this program `make_index.py`.

```
import csv
import cPickle as pickle

def normalise(str):
    return str.strip().lower()

def make_index():
    index = {}
    file = open('dict.csv')
    reader = csv.reader(file, delimiter=',', quotechar='|')
    for row in reader:
```

```
        if len(row) >= 2:
            key = normalise(row[0])
            val = row[1]
            index[key] = val
    file.close()
    return index

index = make_index()
file = open('index.pickle', 'w')
pickle.dump(index, file)
file.close()
```

The `make_index()` function is the same as before. We use the `cPickle` library because it is a lot faster than the ordinary `pickle` version. See the official documentation if you are interested in the details:

http://docs.python.org/library/pickle.html

After executing this program we get a new file called `index.pickle`, which contains a pickled version of the index. It is about the same size as the `dict.csv` file. From now on we can simply load the index from this file, rather than re-building it, thus saving some work.

Exercise 7

Run the `make_index.py` program to create the file `pickle.index`.
Open `pickle.index` in a text editor. What does it look like?

Below is a complete version of `searchme.py` using the pickled index file. The key parts of the program that have changed are highlighted. Notice how the `pickle.load()` function is used to get the index from the pickled file.

```
import csv
import cPickle as pickle

def normalise(str):
    return str.strip().lower()

def get_input():
    prompt = 'Enter a query term: '
    input = raw_input(prompt)
    return input.strip().lower()
```

```
file = open('index.pickle')
index = pickle.load(file)
file.close()

def search(index,query):
    return index.get(normalise(query))

query = get_input()
while len(query) > 0:
   result = search(index,query)
   if result == None:
      print 'Search failed'
   else:
      print result
   query = get_input()
```

Databases and persistent data

In effect, by pickling the index, we have made it *persistent*. Data is persistent if it lives longer than the execution of a single program. For example the documents you make with a word processor are persistent. You save the document to a file when you exit the word processor, and then re-open the document from a file next time you want to edit it. The document survives even though the word processor program is not always running.

Web applications often keep persistent data for their users. In the third stage of the project, that is exactly what you will do, and you will use pickling to do it.

A more sophisticated web application might use a dedicated data base to store persistent information. In fact databases provide both persistence *and* indexing. Thus, what we've really been doing in this lecture is implementing our own little database.

Nowadays companies are providing database services as web applications. The buzzword for this is *cloud computing*. If we have time in the next lecture, we will see how to turn SearchMe into a cloud computing application! Then we will be totally buzzword compliant!