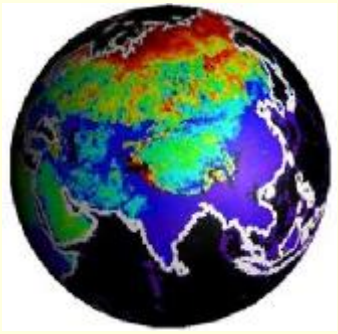

Chapter 13
Case study: data structure selection

Jihn-Fa Jan, Ph.D.

Associate Professor

Department of Land Economics

National Chengchi University

Word frequency analysis

Exercise 13.1 Write a program that reads a file, breaks each line into words, strips whitespace and punctuation from the words, and converts them to lowercase.

Exercise 13.2 Go to Project Gutenberg (gutenberg.net) and download your favorite out-ofcopyright book in plain text format.
Modify your program from the previous exercise to read the book you downloaded, skip over the header information at the beginning of the file, and process the rest of the words as before.
Then modify the program to count the total number of words in the book, and the number of times each word is used.
Print the number of different words used in the book. Compare different books by different authors, written in different eras. Which author uses the most extensive vocabulary?

Exercise 13.3 Modify the program from the previous exercise to print the 20 most frequently-used words in the book.

Exercise 13.4 Modify the previous program to read a word list (see Section 9.1) and then print all the words in the book that are not in the word list. How many of them are typos? How many of them are common words that *should be in the word list, and how many of them are really obscure?*

Random numbers

- Given the same inputs, most computer programs generate the same outputs every time, so they are said to be deterministic.
- Making a program truly nondeterministic turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to use algorithms that generate pseudorandom numbers. Pseudorandom numbers are not truly random because they are generated by a deterministic computation, but just by looking at the numbers it is all but impossible to distinguish them from random.
- The random module provides functions that generate pseudorandom numbers (which I will simply call “random” from here on).

Random numbers (cont'd)

- **The function random returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time you call random, you get the next number in a long series.**

```
import random
for i in range(10):
    x = random.random()
    print x
```

```
>>>
0.258549483281
0.844356068614
0.368185926842
0.84769981248
0.402834570555
0.512147624916
0.431488334661
0.0766477884006
0.835206208695
0.308548295438
```

Random numbers (cont'd)

- The function randint takes parameters low and high and returns an integer between low and high (including both).

```
>>> for i in range(10):
        print random.randint(5,10)


6
10
7
8
5
8
5
5
10
5
>>>
```

Random numbers (cont'd)

- To choose an element from a sequence at random, you can use **choice**:

```
>>> t = [1,2,3,4,5,6,7,8,9,10]
>>> for i in range(10):
        print random.choice(t)


8
5
7
1
4
4
5
9
9
8
>>>
```

```
>>> t = list('ABCDEFGHIJ')
>>> for i in range(10):
        print random.choice(t)


B
H
D
E
I
G
C
B
B
J
>>>
```

Random numbers (cont'd)

- **The randommodule also provides functions to generate random values from continuous distributions including Gaussian, exponential, gamma, and a few more.**

```
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'System
Random', 'TWOPI', 'WichmannHill', '_BuiltinMethodType', '_MethodType', '__all__'
, '__builtins__', '__doc__', '__file__', '__name__', '_acos', '_ceil', '_cos', '
_e', '_exp', '_hexlify', '_inst', '_log', '_pi', '_random', '_sin', '_sqrt', '_t
est', '_test_generator', '_urandom', '_warn', 'betavariate', 'choice', 'expovari
ate', 'gammavariate', 'gauss', 'getrandbits', 'getstate', 'jumpahead', 'lognormv
ariate', 'normalvariate', 'paretovariate', 'randint', 'random', 'randrange', 'sa
mple', 'seed', 'setstate', 'shuffle', 'uniform', 'vonmisesvariate', 'weibullvari
ate']
>>>
```

Word histogram

- **Here is a program that reads a file and builds a histogram of the words in the file:**

```
import string

exclude_str = string.punctuation + string.whitespace

def process_file(filename):
    h = dict()
    fp = open(filename)
    i = 0
    for line in fp:
        #i += 1
        #print 'line #', i
        process_line(line, h)
    return h

def process_line(line, h):
    #exclude_str = string.punctuation + string.whitespace
    line = line.replace('-', ' ')
    for word in line.split():
        word = word.strip(exclude_str)
        word = word.lower()
        h[word] = h.get(word, 0) + 1

def total_words(h):
    return sum(h.values())

def different_words(h):
    return len(h)

hist = process_file('emma.txt')
print 'Total number of words:', total_words(hist)
print 'Number of different words:', different_words(hist)
```

```
>>>
Total number of words: 161073
Number of different words: 7212
>>>
```

Most common words

- To find the most common words, we can apply the DSU pattern; most_common takes a histogram and returns a list of word-frequency tuples, sorted in reverse order by frequency:

```
def most_common(h):
    t = []
    for key, value in h.items():
        t.append((value, key))
    t.sort(reverse=True)
    return t

t = most_common(hist)
print 'The most common words are:'
for freq, word in t[0:10]:
    print word, '\t', freq
```

```
>>>
Total number of words: 161073
Number of different words: 7212
The most common words are:
to      5242
the     5204
and     4897
of      4293
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
>>>
```

Optional parameters

- **We have seen built-in functions and methods that take a variable number of arguments. It is possible to write user-defined functions with optional arguments, too. For example, here is a function that prints the most common words in a histogram**

```
def print_most_common(hist, num=10):
    t = most_common(hist)
    print 'The most common words are:'
    for freq, word in t[0:num]:
        print word, '\t', freq

print_most_common(hist)
print_most_common(hist,5)
```

```
The most common words are:
to      5242
the     5204
and     4897
of      4293
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
The most common words are:
to      5242
the     5204
and     4897
of      4293
i       3191
>>>
```

If a function has both required and optional parameters, all the required parameters have to come first, followed by the optional ones.

Dictionary subtraction

- Finding the words from the book that are not in the word list from words.txt is a problem you might recognize as set subtraction; that is, we want to find all the words from one set (the words in the book) that are not in another set (the words in the list).
- subtract takes dictionaries d1 and d2 and returns a new dictionary that contains all the keys from d1 that are not in d2. Since we don't really care about the values, we set them all to None.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res

words = process_file('words.txt')
diff = subtract(hist, words)
print "The words in the book that aren't in the word list are:"
for word in diff.keys():
    print word,
```

Random words

- **To choose a random word from the histogram, the simplest algorithm is to build a list with multiple copies of each word, according to the observed frequency, and then choose from the list:**

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)
    return random.choice(t)
```

  - The expression [word] * freq creates a list with freq copies of the string word. The extend method is similar to append except that the argument is a sequence.

Markov analysis

- **If you choose words from the book at random, you can get a sense of the vocabulary, you probably won't get a sentence:**

  this the small regard harriet which knightley's it most things

- **A series of random words seldom makes sense because there is no relationship between successive words. For example, in a real sentence you would expect an article like "the" to be followed by an adjective or a noun, and probably not a verb or adverb.**
- **One way to measure these kinds of relationships is Markov analysis, which characterizes, for a given sequence of words, the probability of the word that comes next. For example, the song *Eric, the Half a Bee begins:***

Half a bee, philosophically,
Must, ipso facto, half not be.
But half the bee has got to be
Vis a vis, its entity. D'you see?

But can a bee be said to be
Or not to be an entire bee
When half the bee is not a bee
Due to some ancient injury?

Markov analysis (cont'd)

- In this text, the phrase "half the" is always followed by the word "bee," but the phrase "the bee" might be followed by either "has" or "is".
- The result of Markov analysis is a mapping from each prefix (like "half the" and "the bee") to all possible suffixes (like "has" and "is").
- Given this mapping, you can generate a random text by starting with any prefix and choosing at random from the possible suffixes. Next, you can combine the end of the prefix and the new suffix to form the next prefix, and repeat.

Data structures

- **Using Markov analysis to generate random text is fun, but there is also a point to this exercise: data structure selection. In your solution to the previous exercises, you had to choose:**
  - How to represent the prefixes.
  - How to represent the collection of possible suffixes.
  - How to represent the mapping from each prefix to the collection of possible suffixes.

Debugging

- **When you are debugging a program, and especially if you are working on a hard bug, there are four things to try:**
  - **reading:** Examine your code, read it back to yourself, and check that it says what you meant to say.
  - **running:** Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding.
  - **ruminating:** Take some time to think! What kind of error is it: syntax, runtime, semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?
  - **retreating:** At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can starting rebuilding.