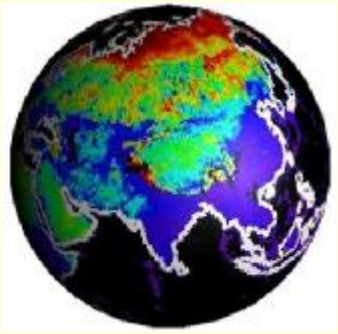

Chapter 9
Case study: Word Play

Jihn-Fa Jan, Ph.D.
Associate Professor
Department of Land Economics
National Chengchi University

Reading word lists

- The test data is a word list in the public domain by Grady Ward. The data contains a list of 113,809 crosswords, which can be downloaded from: http://thinkpython2.com/code/words.txt
- This file is in plain text, so you can open it with a text editor, but you can also read it from Python. The built-in function **open** takes the name of the file as a parameter and returns a **file object** you can use to read the file.

```
>>> fin = open('words.txt')
>>> print(fin)
<_io.TextIOWrapper name='words.txt' mode='r' encoding='cp950'>
```

  - fin is a common name for a file object used for input. Mode 'r' indicates that this file is open for reading (as opposed to 'w' for writing).

Reading word lists

- The file object provides several methods for reading, including readline, which reads characters from the file until it gets to a newline and returns the result as a string:.

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

  - The first word in this particular list is “aa”, which is a kind of lava. The sequence **\r\n** represents two **whitespace characters**, **a carriage return and a newline**, that separate this word from the next.
- The file object keeps track of where it is in the file, so if you call **readline** again, you get the next word:

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

Reading word lists

- We can get rid of the whitespace with the string method strip:

```
>>> line = fin.readline()
>>> word = line.strip()
>>> word
'aahed'
```

- You can also use a file object as part of a for loop. This program reads words.txt and prints each word, one per line:

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

Exercises

Exercise 9.1. *Write a program that reads* `words.txt` *and prints only the words with more than 20 characters (not counting whitespace).*

Exercise 9.2. *In 1939 Ernest Vincent Wright published a 50,000 word novel called* Gadsby *that does not contain the letter "e". Since "e" is the most common letter in English, that's not easy to do.*

In fact, it is difficult to construct a solitary thought without using that most common symbol. It is slow going at first, but with caution and hours of training you can gradually gain facility.

All right, I'll stop now.

Write a function called `has_no_e` *that returns* `True` *if the given word doesn't have the letter "e" in it.*

Modify your program from the previous section to print only the words that have no "e" and compute the percentage of the words in the list that have no "e".

Exercises

Exercise 9.3. *Write a function named* `avoids` *that takes a word and a string of forbidden letters, and that returns* `True` *if the word doesn't use any of the forbidden letters.*

Modify your program to prompt the user to enter a string of forbidden letters and then print the number of words that don't contain any of them. Can you find a combination of 5 forbidden letters that excludes the smallest number of words?

Exercise 9.4. *Write a function named* `uses_only` *that takes a word and a string of letters, and that returns* `True` *if the word contains only letters in the list. Can you make a sentence using only the letters* `acefhlo`*? Other than "Hoe alfalfa?"*

Exercise 9.5. *Write a function named* `uses_all` *that takes a word and a string of required letters, and that returns* `True` *if the word uses all the required letters at least once. How many words are there that use all the vowels* `aeiou`*? How about* `aeiouy`*?*

Exercise 9.6. *Write a function called* `is_abecedarian` *that returns* `True` *if the letters in a word appear in alphabetical order (double letters are ok). How many abecedarian words are there?*

Search

- In Chapter 8.6 Searching

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

- In Chapter 8.7 Looping and counting

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

Search

- Searching a letter in words:

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

  - The for loop traverses the characters in word. If we find the letter “e”, we can immediately return False; otherwise we have to go to the next letter. If we exit the loop normally, that means we didn’t find an “e”, so we return True.
  - You could write this function more concisely using the **in** operator.

Search

- **avoids** is a more general version of **has_no_e** but it has the same structure:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

  - We can return False as soon as we find a forbidden letter; if we get to the end of the loop, we return True.

Search

- Use only available characters:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

  - **uses_only** is similar except that the sense of the condition is reversed. Instead of a list of forbidden words, we have a list of available words. If we find a letter in word that is not in available, we can return False.

Search

- Use the required characters at least once:

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

  - **uses_all** is similar except that we reverse the role of the word and the string of letters. Instead of traversing the letters in word, the loop traverses the required letters. If any of the required letters do not appear in the word, we can return False.

Search

- If you were really thinking like a computer scientist, you would have recognized that **uses_all** was an instance of a previously-solved problem, and you would have written:

```
def uses_all(word, required):
    return uses_only(required, word)
```

- This is an example of a program development plan called **reduction to a previously solved problem**, which means that you recognize the problem you are working on as an instance of a previously-solved problem, and apply a previously-developed solution.

Looping with indices

- **abecedarian**: 照ABC次序的字
- For is_abecedarian we have to compare adjacent letters, which is a little tricky with a for loop:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

Looping with indices

- An alternative is to use recursion:

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

Looping with indices

- Another option is to use a while loop:

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

  - The loop starts at **i=0** and ends when **i=len(word)-1**. Each time through the loop, it compares the ***ith*** character (which you can think of as the current character) to the ***i+1th*** character (which you can think of as the next).
  - If the next character is less than (alphabetically before) the current one, then we have discovered a break in the abecedarian trend, and we return False.

Looping with indices

- 順著寫、倒著寫拼法都一樣的字叫做 **palindrome (回文)**，例如：noon, redivider
- Here is a version of **is_palindrome** (see Exercise 6.3) that uses two indices; one starts at the beginning and goes up; the other starts at the end and goes down.

```
def is_palindrome(word):
    i = 0
    j = len(word)-1

    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1

    return True
```

Debugging

- Testing programs is hard. The functions in this chapter are relatively easy to test because you can check the results by hand. Even so, it is somewhere between difficult and impossible to choose a set of words that test for all possible errors.
- In general, testing can help you find bugs, but it is not easy to generate a good set of test cases, and even if you do, you can't be sure your program is correct.
- According to a legendary computer scientist:

  Program testing can be used to show the presence of bugs, but never to show their absence!

  — Edsger W. Dijkstra