
Case Study: Data Structure Selection

At this point you have learned about Python's core data structures, and you have seen some of the algorithms that use them. If you would like to know more about algorithms, this might be a good time to read [Chapter 21](#). But you don't have to read it before you go on; you can read it whenever you are interested.

This chapter presents a case study with exercises that let you think about choosing data structures and practice using them.

Word Frequency Analysis

As usual, you should at least attempt the exercises before you read my solutions.

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.

Hint: The `string` module provides a string named `whitespace`, which contains space, tab, newline, etc., and `punctuation` which contains the punctuation characters. Let's see if we can make Python swear:

```
>>> import string
>>> string.punctuation
'!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Also, you might consider using the string methods `strip`, `replace` and `translate`.

Exercise 13-2.

Go to Project Gutenberg (<http://gutenberg.org>) and download your favorite out-of-copyright 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 “[Reading Word Lists](#)” on page 99) 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**. Determinism is usually a good thing, since we expect the same calculation to yield the same result. For some applications, though, we want the computer to be unpredictable. Games are an obvious example, but there are more.

Making a program truly nondeterministic turns out to be difficult, 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).

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. To see a sample, run this loop:

```
import random

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

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

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

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

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

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

Exercise 13-5.

Write a function named `choose_from_hist` that takes a histogram as defined in “**Dictionary as a Collection of Counters**” on page 127 and returns a random value from the histogram, chosen with probability in proportion to frequency. For example, for this histogram:

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

your function should return `'a'` with probability $2/3$ and `'b'` with probability $1/3$.

Word Histogram

You should attempt the previous exercises before you go on. You can download my solution from http://thinkpython2.com/code/analyze_book1.py. You will also need <http://thinkpython2.com/code/emma.txt>.

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

```

import string

def process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, hist)
    return hist

def process_line(line, hist):
    line = line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()
        hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')

```

This program reads `emma.txt`, which contains the text of *Emma* by Jane Austen.

`process_file` loops through the lines of the file, passing them one at a time to `process_line`. The histogram `hist` is being used as an accumulator.

`process_line` uses the string method `replace` to replace hyphens with spaces before using `split` to break the line into a list of strings. It traverses the list of words and uses `strip` and `lower` to remove punctuation and convert to lowercase. (It is shorthand to say that strings are “converted”; remember that strings are immutable, so methods like `strip` and `lower` return new strings.)

Finally, `process_line` updates the histogram by creating a new item or incrementing an existing one.

To count the total number of words in the file, we can add up the frequencies in the histogram:

```

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

```

The number of different words is just the number of items in the dictionary:

```

def different_words(hist):
    return len(hist)

```

Here is some code to print the results:

```

print('Total number of words:', total_words(hist))
print('Number of different words:', different_words(hist))

```

And the results:

```

Total number of words: 161080
Number of different words: 7214

```

Most Common Words

To find the most common words, we can make a list of tuples, where each tuple contains a word and its frequency, and sort it.

The following function takes a histogram and returns a list of word-frequency tuples:

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

In each tuple, the frequency appears first, so the resulting list is sorted by frequency. Here is a loop that prints the 10 most common words:

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

I use the keyword argument `sep` to tell `print` to use a tab character as a “separator”, rather than a space, so the second column is lined up. Here are the results from *Emma*:

```
The most common words are:
to      5242
the     5205
and     4897
of      4295
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

This code can be simplified using the `key` parameter of the `sort` function. If you are curious, you can read about it at <https://wiki.python.org/moin/HowTo/Sorting>.

Optional Parameters

We have seen built-in functions and methods that take optional arguments. It is possible to write programmer-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[:num]:
        print(word, freq, sep='\t')
```

The first parameter is required; the second is optional. The **default value** of num is 10.

If you only provide one argument:

```
print_most_common(hist)
```

num gets the default value. If you provide two arguments:

```
print_most_common(hist, 20)
```

num gets the value of the argument instead. In other words, the optional argument **overrides** the default value.

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 the other (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
```

To find the words in the book that are not in words.txt, we can use process_file to build a histogram for words.txt, and then subtract:

```
words = process_file('words.txt')
diff = subtract(hist, words)

print("Words in the book that aren't in the word list:")
for word in diff:
    print(word, end=' ')
```

Here are some of the results from *Emma*:

```
Words in the book that aren't in the word list:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

Some of these words are names and possessives. Others, like “rencontre”, are no longer in common use. But a few are common words that should really be in the list!

Exercise 13-6.

Python provides a data structure called `set` that provides many common set operations. You can read about them in “Sets” on page 226, or read the documentation at <http://docs.python.org/3/library/stdtypes.html#types-set>.

Write a program that uses set subtraction to find words in the book that are not in the word list.

Solution: http://thinkpython2.com/code/analyze_book2.py.

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.

This algorithm works, but it is not very efficient; each time you choose a random word, it rebuilds the list, which is as big as the original book. An obvious improvement is to build the list once and then make multiple selections, but the list is still big.

An alternative is:

1. Use keys to get a list of the words in the book.
2. Build a list that contains the cumulative sum of the word frequencies (see [Exercise 10-2](#)). The last item in this list is the total number of words in the book, n .

3. Choose a random number from 1 to n . Use a bisection search (See [Exercise 10-10](#)) to find the index where the random number would be inserted in the cumulative sum.
4. Use the index to find the corresponding word in the word list.

Exercise 13-7.

Write a program that uses this algorithm to choose a random word from the book.

Solution: http://thinkpython2.com/code/analyze_book3.py.

Markov Analysis

If you choose words from the book at random, you can get a sense of the vocabulary, but 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 words that might come 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?

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.

For example, if you start with the prefix “Half a”, then the next word has to be “bee”, because the prefix only appears once in the text. The next prefix is “a bee”, so the next suffix might be “philosophically”, “be” or “due”.

In this example the length of the prefix is always two, but you can do Markov analysis with any prefix length.

Exercise 13-8.

Markov analysis:

1. Write a program to read a text from a file and perform Markov analysis. The result should be a dictionary that maps from prefixes to a collection of possible suffixes. The collection might be a list, tuple, or dictionary; it is up to you to make an appropriate choice. You can test your program with prefix length 2, but you should write the program in a way that makes it easy to try other lengths.
2. Add a function to the previous program to generate random text based on the Markov analysis. Here is an example from *Emma* with prefix length 2:

He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were never meant for me?” “I cannot make speeches, Emma:” he soon cut it all himself.

For this example, I left the punctuation attached to the words. The result is almost syntactically correct, but not quite. Semantically, it almost makes sense, but not quite.

What happens if you increase the prefix length? Does the random text make more sense?

3. Once your program is working, you might want to try a mash-up: if you combine text from two or more books, the random text you generate will blend the vocabulary and phrases from the sources in interesting ways.

Credit: This case study is based on an example from Kernighan and Pike, *The Practice of Programming*, Addison-Wesley, 1999.

You should attempt this exercise before you go on; then you can download my solution from <http://thinkpython2.com/code/markov.py>. You will also need <http://thinkpython2.com/code/emma.txt>.

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.

The last one is easy: a dictionary is the obvious choice for a mapping from keys to corresponding values.

For the prefixes, the most obvious options are string, list of strings, or tuple of strings.

For the suffixes, one option is a list; another is a histogram (dictionary).

How should you choose? The first step is to think about the operations you will need to implement for each data structure. For the prefixes, we need to be able to remove words from the beginning and add to the end. For example, if the current prefix is “Half a”, and the next word is “bee”, you need to be able to form the next prefix, “a bee”.

Your first choice might be a list, since it is easy to add and remove elements, but we also need to be able to use the prefixes as keys in a dictionary, so that rules out lists. With tuples, you can’t append or remove, but you can use the addition operator to form a new tuple:

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

`shift` takes a tuple of words, `prefix`, and a string, `word`, and forms a new tuple that has all the words in `prefix` except the first, and `word` added to the end.

For the collection of suffixes, the operations we need to perform include adding a new suffix (or increasing the frequency of an existing one), and choosing a random suffix.

Adding a new suffix is equally easy for the list implementation or the histogram. Choosing a random element from a list is easy; choosing from a histogram is harder to do efficiently (see [Exercise 13-7](#)).

So far we have been talking mostly about ease of implementation, but there are other factors to consider in choosing data structures. One is runtime. Sometimes there is a theoretical reason to expect one data structure to be faster than other; for example, I mentioned that the `in` operator is faster for dictionaries than for lists, at least when the number of elements is large.

But often you don’t know ahead of time which implementation will be faster. One option is to implement both of them and see which is better. This approach is called **benchmarking**. A practical alternative is to choose the data structure that is easiest to implement, and then see if it is fast enough for the intended application. If so, there is

no need to go on. If not, there are tools, like the `profile` module, that can identify the places in a program that take the most time.

The other factor to consider is storage space. For example, using a histogram for the collection of suffixes might take less space because you only have to store each word once, no matter how many times it appears in the text. In some cases, saving space can also make your program run faster, and in the extreme, your program might not run at all if you run out of memory. But for many applications, space is a secondary consideration after runtime.

One final thought: in this discussion, I have implied that we should use one data structure for both analysis and generation. But since these are separate phases, it would also be possible to use one structure for analysis and then convert to another structure for generation. This would be a net win if the time saved during generation exceeded the time spent in conversion.

Debugging

When you are debugging a program, and especially if you are working on a hard bug, there are five 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 build scaffolding.

Ruminating:

Take some time to think! What kind of error is it: syntax, runtime, or 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?

Rubberducking:

If you explain the problem to someone else, you sometimes find the answer before you finish asking the question. Often you don't need the other person; you could just talk to a rubber duck. And that's the origin of the well-known strategy called **rubber duck debugging**. I am not making this up; see https://en.wikipedia.org/wiki/Rubber_duck_debugging.

Retreating:

At some point, the best thing to do is back off and undo recent changes until you get back to a program that works and that you understand. Then you can start rebuilding.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call "random walk programming", which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works and that you understand.

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can copy the pieces back one at a time.

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

Glossary

deterministic:

Pertaining to a program that does the same thing each time it runs, given the same inputs.

pseudorandom:

Pertaining to a sequence of numbers that appears to be random, but is generated by a deterministic program.

default value:

The value given to an optional parameter if no argument is provided.

override:

To replace a default value with an argument.

benchmarking:

The process of choosing between data structures by implementing alternatives and testing them on a sample of the possible inputs.

rubber duck debugging:

Debugging by explaining your problem to an inanimate object such as a rubber duck. Articulating the problem can help you solve it, even if the rubber duck doesn't know Python.

Exercises

Exercise 13-9.

The “rank” of a word is its position in a list of words sorted by frequency: the most common word has rank 1, the second most common has rank 2, etc.

Zipf's law describes a relationship between the ranks and frequencies of words in natural languages (http://en.wikipedia.org/wiki/Zipf's_law). Specifically, it predicts that the frequency, f , of the word with rank r is:

$$f = cr^{-s}$$

where s and c are parameters that depend on the language and the text. If you take the logarithm of both sides of this equation, you get:

$$\log f = \log c - s \log r$$

So if you plot $\log f$ versus $\log r$, you should get a straight line with slope $-s$ and intercept $\log c$.

Write a program that reads a text from a file, counts word frequencies, and prints one line for each word, in descending order of frequency, with $\log f$ and $\log r$. Use the graphing program of your choice to plot the results and check whether they form a straight line. Can you estimate the value of s ?

Solution: <http://thinkpython2.com/code/zipf.py>. To run my solution, you need the plotting module `matplotlib`. If you installed Anaconda, you already have `matplotlib`; otherwise you might have to install it.