**Chapter 9**

Dictionaries

A *dictionary* is like a list, but more general. In a list, the index positions have to be integers; in a dictionary, the indices can be (almost) any type.

You can think of a dictionary as a mapping between a set of indices (which are called *keys*) and a set of values. Each key maps to a value. The association of a key and a value is called a *key-value pair* or sometimes an *item*.

As an example, we’ll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.



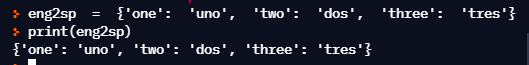
The curly brackets, {}, represent an empty dictionary. To add items to the dictio- nary, you can use square brackets:



This line creates an item that maps from the key 'one' to the value “uno”. If we print the dictionary again, we see a key-value pair with a colon between the key and value:



This output format is also an input format. For example, you can create a new dictionary with three items. But if you print eng2sp, you might be surprised:



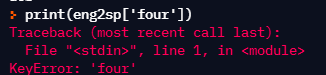
The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

But that’s not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:



The key 'two' always maps to the value “dos” so the order of the items doesn’t matter.

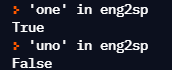
If the key isn’t in the dictionary, you get an exception:



The len function works on dictionaries; it returns the number of key-value pairs:



The in operator works on dictionaries; it tells you whether something appears as a *key* in the dictionary (appearing as a value is not good enough).



To see whether something appears as a value in a dictionary, you can use the method values, which returns the values as a type that can be converted to a list, and then use the in operator:



The in operator uses different algorithms for lists and dictionaries. For lists, it uses a linear search algorithm. As the list gets longer, the search time gets longer in direct proportion to the length of the list. For dictionaries, Python uses an algorithm called a *hash table* that has a remarkable property: the in operator takes about the same amount of time no matter how many items there are in a dictionary. I won’t explain why hash functions are so magical, but you can read more about it at wikipedia.org/wiki/Hash\_table.

**Exercise 1: Download a copy of the file** [www.py4e.com/code3/words.txt](http://www.py4e.com/code3/words.txt)

## Write a program that reads the words in *words.txt* and stores them as keys in a dictionary. It doesn’t matter what the values are. Then you can use the in operator as a fast way to check whether a string is in the dictionary.

# Dictionary as a set of counters

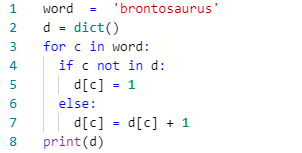
Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

* + 1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
    2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.
    3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An *implementation* is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementa- tion is that we don’t have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear.

Here is what the code might look like:



We are effectively computing a *histogram*, which is a statistical term for a set of counters (or frequencies).

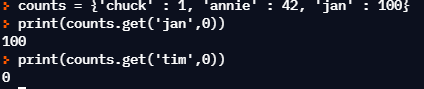
The for loop traverses the string. Each time through the loop, if the character c is not in the dictionary, we create a new item with key c and the initial value 1 (since we have seen this letter once). If c is already in the dictionary we increment d[c].

Here’s the output of the program:

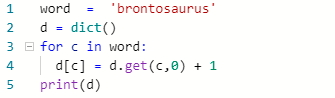
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}

The histogram indicates that the letters “a” and “b” appear once; “o” appears twice, and so on.

Dictionaries have a method called get that takes a key and a default value. If the key appears in the dictionary, get returns the corresponding value; otherwise it returns the default value. For example:



We can use get to write our histogram loop more concisely. Because the get method automatically handles the case where a key is not in a dictionary, we can reduce four lines down to one and eliminate the if statement.



The use of the get method to simplify this counting loop ends up being a very commonly used “idiom” in Python and we will use it many times in the rest of the book. So you should take a moment and compare the loop using the if statement and in operator with the loop using the get method. They do exactly the same thing, but one is more succinct.

# Dictionaries and files

One of the common uses of a dictionary is to count the occurrence of words in a file with some written text. Let’s start with a very simple file of words taken from the text of *Romeo and Juliet*.

For the first set of examples, we will use a shortened and simplified version of the text with no punctuation. Later we will work with the text of the scene with punctuation included.

But soft what light through yonder window breaks It is the east and Juliet is the sun

Arise fair sun and kill the envious moon Who is already sick and pale with grief

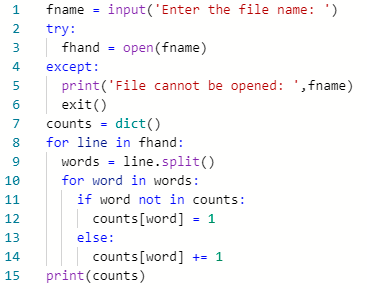
We will write a Python program to read through the lines of the file, break each line into a list of words, and then loop through each of the words in the line and count each word using a dictionary.

You will see that we have two for loops. The outer loop is reading the lines of the file and the inner loop is iterating through each of the words on that particular line. This is an example of a pattern called *nested loops* because one of the loops is the *outer* loop and the other loop is the *inner* loop.

Because the inner loop executes all of its iterations each time the outer loop makes a single iteration, we think of the inner loop as iterating “more quickly” and the outer loop as iterating more slowly.

The combination of the two nested loops ensures that we will count every word on every line of the input file.

*# Code:* [*http://www.py4e.com/code3/count1.py*](http://www.py4e.com/code3/count1.py)

**

In our else statement, we use the more compact alternative for incrementing a variable. counts[word] += 1 is equivalent to counts[word] = counts[word] + 1. Either method can be used to change the value of a variable by any desired amount. Similar alternatives exist for -=, \*=, and /=.

When we run the program, we see a raw dump of all of the counts in unsorted hash order. (the *romeo.txt* file is available at [www.py4e.com/code3/romeo.txt)](http://www.py4e.com/code3/romeo.txt))

python count1.py

Enter the file name: romeo.txt

{'and': 3, 'envious': 1, 'already': 1, 'fair': 1,

'is': 3, 'through': 1, 'pale': 1, 'yonder': 1,

'what': 1, 'sun': 2, 'Who': 1, 'But': 1, 'moon': 1,

'window': 1, 'sick': 1, 'east': 1, 'breaks': 1,

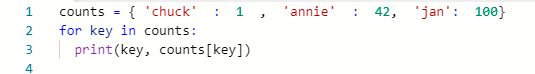
'grief': 1, 'with': 1, 'light': 1, 'It': 1, 'Arise': 1,

'kill': 1, 'the': 3, 'soft': 1, 'Juliet': 1}

It is a bit inconvenient to look through the dictionary to find the most common words and their counts, so we need to add some more Python code to get us the output that will be more helpful.

# Looping and dictionaries

If you use a dictionary as the sequence in a for statement, it traverses the keys of the dictionary. This loop prints each key and the corresponding value:



Here’s what the output looks like:

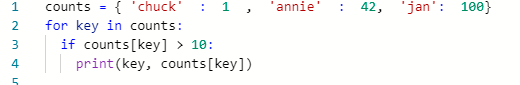
jan 100

chuck 1

annie 42

Again, the keys are in no particular order.

We can use this pattern to implement the various loop idioms that we have de- scribed earlier. For example if we wanted to find all the entries in a dictionary with a value above ten, we could write the following code:



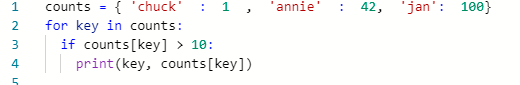
The for loop iterates through the *keys* of the dictionary, so we must use the index operator to retrieve the corresponding *value* for each key. Here’s what the output looks like:

jan 100

annie 42

We see only the entries with a value above 10.

If you want to print the keys in alphabetical order, you first make a list of the keys in the dictionary using the keys method available in dictionary objects, and then sort that list and loop through the sorted list, looking up each key and printing out key-value pairs in sorted order as follows:



Here’s what the output looks like:

['jan', 'chuck', 'annie'] annie 42

chuck 1

jan 100

First you see the list of keys in unsorted order that we get from the keys method. Then we see the key-value pairs in order from the for loop.

# Advanced text parsing

In the above example using the file *romeo.txt*, we made the file as simple as possible by removing all punctuation by hand. The actual text has lots of punctuation, as shown below.

But, soft! what light through yonder window breaks? It is the east, and Juliet is the sun.

Arise, fair sun, and kill the envious moon, Who is already sick and pale with grief,

Since the Python split function looks for spaces and treats words as tokens sep- arated by spaces, we would treat the words “soft!” and “soft” as *different* words and create a separate dictionary entry for each word.

Also since the file has capitalization, we would treat “who” and “Who” as different words with different counts.

We can solve both these problems by using the string methods lower, punctuation, and translate. The translate is the most subtle of the methods. Here is the documentation for translate:

line.translate(str.maketrans(fromstr, tostr, deletestr))

*Replace the characters in fromstr with the character in the same position in tostr and delete all characters that are in deletestr. The fromstr and tostr can be empty strings and the deletestr parameter can be omitted.*

We will not specify the tostr but we will use the deletestr parameter to delete all of the punctuation. We will even let Python tell us the list of characters that it considers “punctuation”:

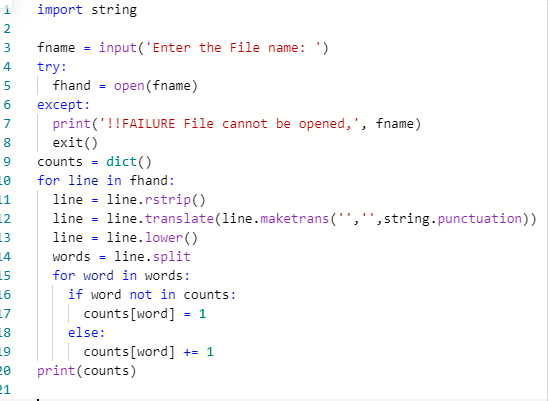
>>> import string

>>> string.punctuation

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

The parameters used by translate were different in Python 2.0. We make the following modifications to our program:

import string



*# Code:* [*http://www.py4e.com/code3/count2.py*](http://www.py4e.com/code3/count2.py)

Part of learning the “Art of Python” or “Thinking Pythonically” is realizing that Python often has built-in capabilities for many common data analysis problems. Over time, you will see enough example code and read enough of the documentation to know where to look to see if someone has already written something that makes your job much easier.

The following is an abbreviated version of the output:

Enter the file name: romeo-full.txt

{'swearst': 1, 'all': 6, 'afeard': 1, 'leave': 2, 'these': 2,

'kinsmen': 2, 'what': 11, 'thinkst': 1, 'love': 24, 'cloak': 1,

a': 24, 'orchard': 2, 'light': 5, 'lovers': 2, 'romeo': 40,

'maiden': 1, 'whiteupturned': 1, 'juliet': 32, 'gentleman': 1,

'it': 22, 'leans': 1, 'canst': 1, 'having': 1, ...}

Looking through this output is still unwieldy and we can use Python to give us exactly what we are looking for, but to do so, we need to learn about Python *tuples*. We will pick up this example once we learn about tuples.

# Debugging

As you work with bigger datasets it can become unwieldy to debug by printing and checking data by hand. Here are some suggestions for debugging large datasets:

**Scale down the input** If possible, reduce the size of the dataset. For example if the program reads a text file, start with just the first 10 lines, or with the smallest example you can find. You can either edit the files themselves, or (better) modify the program so it reads only the first n lines.

If there is an error, you can reduce n to the smallest value that manifests the error, and then increase it gradually as you find and correct errors.

**Check summaries and types** Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers.

A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.

**Write self-checks** Sometimes you can write code to check for errors automati- cally. For example, if you are computing the average of a list of numbers, you could check that the result is not greater than the largest element in the list or less than the smallest. This is called a “sanity check” because it detects results that are “completely illogical”. Another kind of check compares the results of two different computations to see if they are consistent. This is called a “consistency check”.

**Pretty print the output** Formatting debugging output can make it easier to spot an error.

Again, time you spend building scaffolding can reduce the time you spend debug- ging.

# Glossary

**dictionary** A mapping from a set of keys to their corresponding values.

**hashtable** The algorithm used to implement Python dictionaries.

**hash function** A function used by a hashtable to compute the location for a key.

**histogram** A set of counters.

**implementation** A way of performing a computation.

**item** Another name for a key-value pair.

**key** An object that appears in a dictionary as the first part of a key-value pair.

**key-value pair** The representation of the mapping from a key to a value.

**lookup** A dictionary operation that takes a key and finds the corresponding value.

**nested loops** When there are one or more loops “inside” of another loop. The inner loop runs to completion each time the outer loop runs once.

**value** An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word “value”.

# Exercises

## Exercise 2: Write a program that categorizes each mail message by which day of the week the commit was done. To do this look for lines that start with “From”, then look for the third word and keep a running count of each of the days of the week. At the end of the program print out the contents of your dictionary (order does not matter).

**Sample Line:**

From [stephen.marquard@uct.ac.za](mailto:stephen.marquard@uct.ac.za) Sat Jan 5 09:14:16 2008

## Sample Execution:

python dow.py

Enter a file name: mbox-short.txt

{'Fri': 20, 'Thu': 6, 'Sat': 1}

## Exercise 3: Write a program to read through a mail log, build a his- togram using a dictionary to count how many messages have come from each email address, and print the dictionary.

Enter file name: mbox-short.txt

{'gopal.ramasammycook@gmail.com': 1, 'louis@media.berkeley.edu': 3,

'cwen@iupui.edu': 5, 'antranig@caret.cam.ac.uk': 1,

'rjlowe@iupui.edu': 2, 'gsilver@umich.edu': 3,

'david.horwitz@uct.ac.za': 4, 'wagnermr@iupui.edu': 1,

'zqian@umich.edu': 4, 'stephen.marquard@uct.ac.za': 2,

'ray@media.berkeley.edu': 1}

## Exercise 4: Add code to the above program to figure out who has the most messages in the file. After all the data has been read and the dic- tionary has been created, look through the dictionary using a maximum loop (see Chapter 5: Maximum and minimum loops) to find who has the most messages and print how many messages the person has.

Enter a file name: mbox-short.txt [cwen@iupui.edu](mailto:cwen@iupui.edu) 5

Enter a file name: mbox.txt [zqian@umich.edu](mailto:zqian@umich.edu) 195

## Exercise 5: This program records the domain name (instead of the address) where the message was sent from instead of who the mail came from (i.e., the whole email address). At the end of the program, print out the contents of your dictionary.

python schoolcount.py

Enter a file name: mbox-short.txt

{'media.berkeley.edu': 4, 'uct.ac.za': 6, 'umich.edu': 7,

'gmail.com': 1, 'caret.cam.ac.uk': 1, 'iupui.edu': 8}