Chapter 11

Dictionaries

A dictionary is a more general version of a list. Here is a list that contains the number of days in the months of the year:

```
days = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

If we want the the number of days in January, use `days[0]`. December is `days[11]` or `days[-1]`.

Here is a dictionary of the days in the months of the year:

```
days = {'January':31, 'February':28, 'March':31, 'April':30,
        'May':31, 'June':30, 'July':31, 'August':31,
        'September':30, 'October':31, 'November':30, 'December':31}
```

To get the number of days in January, we use `days['January']`. One benefit of using dictionaries here is the code is more readable, and we don't have to figure out which index in the list a given month is at. Dictionaries have a number of other uses, as well.

11.1 Basics

Creating dictionaries Here is a simple dictionary:

```
d = {'A':100, 'B':200}
```

To declare a dictionary we enclose it in curly braces, `{}`. Each entry consists of a pair separated by a colon. The first part of the pair is called the *key* and the second is the *value*. The key acts like an index. So in the first pair, `'A':100`, the key is `'A'`, the value is `100`, and `d['A']` gives `100`. Keys are often strings, but they can be integers, floats, and many other things as well. You can mix different types of keys in the same dictionary and different types of values, too.

Changing dictionaries Let's start with this dictionary:

```
d = {'A':100, 'B':200}
```

- To change `d['A']` to 400, do

```
d['A']=400
```

- To add a new entry to the dictionary, we can just assign it, like below:

```
d['C']=500
```

  Note that this sort of thing does not work with lists. Doing `L[2]=500` on a list with two elements would produce an index out of range error. But it does work with dictionaries.

- To delete an entry from a dictionary, use the **`del`** operator:

```
del d['A']
```

Empty dictionary The empty dictionary is `{}`, which is the dictionary equivalent of `[]` for lists or `''` for strings.

Important note The order of items in a dictionary will not necessarily be the order in which put them into the dictionary. Internally, Python rearranges things in a dictionary in order to optimize performance.

11.2 Dictionary examples

Example 1 You can use a dictionary as an actual dictionary of definitions:

```
d = {'dog' : 'has a tail and goes woof!',
     'cat' : 'says meow',
     'mouse' : 'chased by cats'}
```

Here is an example of the dictionary in use:

```
word = input('Enter a word: ')
print('The definition is:', d[word])
```

```
Enter a word: mouse
The definition is: chased by cats
```

Example 2 The following dictionary is useful in a program that works with Roman numerals.

```
numerals = {'I':1, 'V':5, 'X':10, 'L':50, 'C':100, 'D':500, 'M':1000}
```

Example 3 In the game Scrabble, each letter has a point value associated with it. We can use the following dictionary for the letter values:

```
points = {'A':1, 'B':3, 'C':3, 'D':2, 'E':1, 'F':4, 'G':2,
          'H':4, 'I':1, 'J':8, 'K':5, 'L':1, 'M':3, 'N':1,
          'O':1, 'P':3, 'Q':10, 'R':1, 'S':1, 'T':1, 'U':1,
          'V':4, 'W':4, 'X':8, 'Y':4, 'Z':10}
```

To score a word, we can do the following:

```
score = sum([points[c] for c in word])
```

Or, if you prefer the long way:

```
total = 0
for c in word:
    total += points[c]
```

Example 4 A dictionary provides a nice way to represent a deck of cards:

```
deck = [{'value':i, 'suit':c}
        for c in ['spades', 'clubs', 'hearts', 'diamonds']
        for i in range(2,15)]
```

The deck is actually a list of 52 dictionaries. The `shuffle` method can be used to shuffle the deck:

```
shuffle(deck)
```

The first card in the deck is `deck[0]`. To get the value and the suit of the card, we would use the following:

```
deck[0]['value']
deck[0]['suit']
```

11.3 Working with dictionaries

Copying dictionaries Just like for lists, making copies of dictionaries is a little tricky for reasons we will cover later. To copy a dictionary, use its `copy` method. Here is an example:

```
d2 = d.copy()
```

`in` The `in` operator is used to tell if something is a key in the dictionary. For instance, say we have the following dictionary:

```
d = {'A':100, 'B':200}
```

Referring to a key that is not in the dictionary will produce an error. For instance, `print(d['C'])` will fail. To prevent this error, we can use the `in` operator to check first if a key is in the dictionary before trying to use the key. Here is an example:

```
letter = input('Enter a letter: ')
if letter in d:
    print('The value is', d[letter])
else:
    print('Not in dictionary')
```

You can also use `not in` to see if a key is not in the dictionary.

Looping Looping through dictionaries is similar to looping through lists. Here is an example that prints the keys in a dictionary:

```
for key in d:
    print(key)
```

Here is an example that prints the values:

```
for key in d:
    print(d[key])
```

Lists of keys and values The following table illustrates the ways to get lists of keys and values from a dictionary. It uses the dictionary `d={'A':1,'B':3}`.

Statement	Result	Description
`list(d)`	`['A','B']`	keys of `d`
`list(d.values())`	`[1,3]`	values of `d`
`list(d.items())`	`[('A',1),('B',3)]`	(key,value) pairs of `d`

The pairs returned by `d.items` are called *tuples*. Tuples are a lot like lists. They are covered in Section 19.2.

Here is a use of `d.items` to find all the keys in a dictionary `d` that correspond to a value of 100:

```
d = {'A':100, 'B':200, 'C':100}
L = [x[0] for x in d.items() if x[1]==100]
```

```
['A', 'C']
```

`dict` The `dict` function is another way to create a dictionary. One use for it is kind of like the opposite of the `items` method:

```
d = dict([('A',100),('B',300)])
```

This creates the dictionary `{'A':100,'B':300}`. This way of building a dictionary is useful if your program needs to construct a dictionary while it is running.

Dictionary comprehensions Dictionary comprehensions work similarly to list comprehensions. The following simple example creates a dictionary from a list of words, where the values are the lengths of the words:

```
d = {s : len(s) for s in words}
```

11.4 Counting words

We can use dictionaries to count how frequently certain words appear in a text.

In Section 12.1, we will learn how to read from a text file. For now, here's a line of code that reads the entire contents of a file containing the text of Shakespeare's *Romeo and Juliet* and stores the contents in a string called `text`:

```
text = open('romeoandjuliet.txt').read()
```

To get at the individual words, we will use the `split` method to turn the string into a list of its individual words. Also, because some words may be capitalized, we will convert the whole string to lowercase. We also have to remove punctuation.

```
from string import punctuation

text = text.lower()
for p in punctuation:
    text = text.replace(p, '')
words = text.split()
```

Next comes the dictionary code that does the counting. The dictionary keys will be the words from the text and the values will be counts of how many time each word appears. We start with an empty dictionary. Then for every word in the list of words, if we have seen the word before, we add one to its count, and otherwise we set the count for that word equal to 1. Here is the code:

```
d = {}
for w in words:
    if w in d:
        d[w] = d[w] + 1
    else:
        d[w] = 1
```

Once we have created the dictionary, we can use the following code to print the items in alphabetical order:

```
items = list(d.items())
items.sort()
for i in items:
    print(i)
```

The way this works is a little tricky. Remember that `d.items()` returns a list of pairs (called tuples), which are a lot like lists. When we sort a list of tuples, the sorting is done by the first entry, which in this case is the word. So the sorting is done alphabetically.

If we instead want to order things by frequency, we can flip the order of the tuples and then sort:

```
items = list(d.items())
items = [(i[1], i[0]) for i in items]
items.sort()
for i in items:
    print(i)
```

Here is the code all together:

```
from string import punctuation

# read from file, remove caps and punctuation, and split into words
text = open('romeoandjuliet.txt').read()
```

```
text = text.lower()
for p in punctuation:
    text = text.replace(p, '')
words = text.split()

# build the dictionary of frequencies
d = {}
for w in words:
    if w in d:
        d[w] = d[w] + 1
    else:
        d[w] = 1

# print in alphabetical order
items = list(d.items())
items.sort()
for i in items:
    print(i)

# print in order from least to most common
items = list(d.items())
items = [(i[1], i[0]) for i in items]
items.sort()
for i in items:
    print(i)
```

See Section 24.5 for another approach to word frequencies.

11.5 Exercises

1. Write a program that repeatedly asks the user to enter product names and prices. Store all of these in a dictionary whose keys are the product names and whose values are the prices. When the user is done entering products and prices, allow them to repeatedly enter a product name and print the corresponding price or a message if the product is not in the dictionary.

2. Using the dictionary created in the previous problem, allow the user to enter a dollar amount and print out all the products whose price is less than that amount.

3. For this problem, use the dictionary from the beginning of this chapter whose keys are month names and whose values are the number of days in the corresponding months.

   (a) Ask the user to enter a month name and use the dictionary to tell them how many days are in the month.

   (b) Print out all of the keys in alphabetical order.

   (c) Print out all of the months with 31 days.

   (d) Print out the (key-value) pairs sorted by the number of days in each month

(e) Modify the program from part (a) and the dictionary so that the user does not have to know how to spell the month name exactly. That is, all they have to do is spell the first three letters of the month name correctly.

4. Write a program that uses a dictionary that contains ten user names and passwords. The program should ask the user to enter their username and password. If the username is not in the dictionary, the program should indicate that the person is not a valid user of the system. If the username is in the dictionary, but the user does not enter the right password, the program should say that the password is invalid. If the password is correct, then the program should tell the user that they are now logged in to the system.

5. Repeatedly ask the user to enter a team name and the how many games the team won and how many they lost. Store this information in a dictionary where the keys are the team names and the values are lists of the form `[`*wins*`,` *losses*`]`.

   (a) Using the dictionary created above, allow the user to enter a team name and print out the team's winning percentage.

   (b) Using the dictionary, create a list whose entries are the number of wins of each team.

   (c) Using the dictionary, create a list of all those teams that have winning records.

6. Repeatedly ask the user to enter game scores in a format like *team1 score1 - team2 score2*. Store this information in a dictionary where the keys are the team names and the values are lists of the form `[`*wins*`,` *losses*`]`.

7. Create a 5×5 list of numbers. Then write a program that creates a dictionary whose keys are the numbers and whose values are the how many times the number occurs. Then print the three most common numbers.

8. Using the card dictionary from earlier in this chapter, create a simple card game that deals two players three cards each. The player with the highest card wins. If there is a tie, then compare the second highest card and, if necessary, the third highest. If all three cards have the same value, then the game is a draw.

9. Using the card dictionary from earlier in the chapter, deal out three cards. Determine the following:

   (a) If the three cards form a flush (all of the same suit)

   (b) If there is a three-of-a-kind (all of the same value)

   (c) If there is a pair, but not three-of-a-kind

   (d) If the three cards form a straight (all in a row, like (2, 3, 4) or (10, Jack, Queen))

10. Using the card dictionary from earlier in the chapter run a Monte Carlo simulation to estimate the probability of being dealt a flush in a five card hand. See Exercise 32 of Chapter 10 for more about Monte Carlo simulations.

11. In Section 6.10 we met the substitution cipher. This cipher replaces every letter with a different letter. For instance every *a* might be replaced with an *e*, every *b* might be replaced with an

a, etc. Write a program that asks the user to enter two strings. Then determine if the second string could be an encoded version of the first one with a substitution cipher. For instance, CXYZ is not an encoded version of BOOK because O got mapped to two separate letters. Also, CXXK is not an encoded version of BOOK, because K got mapped to itself. On the other hand, CXXZ would be an encoding of BOOK. This problem can be done with or without a dictionary.

12. Below are the notes used in music:

```
C C# D D# E F F# G G# A A# B
```

The notes for the C major chord are C, E, G. A mathematical way to get this is that E is 4 steps past C and G is 7 steps past C. This works for any base. For example, the notes for D major are D, F#, A. We can represent the major chord steps as a list with two elements: `[4,7]`. The corresponding lists for some other chord types are shown below:

Minor	`[3,7]`	Dominant seventh	`[4,7,10]`
Augmented fifth	`[4,8]`	Minor seventh	`[3,7,10]`
Minor fifth	`[4,6]`	Major seventh	`[4,7,11]`
Major sixth	`[4,7,9]`	Diminished seventh	`[3,6,10]`
Minor sixth	`[3,7,9]`		

Write a program that asks the user for the key and the chord type and prints out the notes of the chord. Use a dictionary whose keys are the (musical) keys and whose values are the lists of steps.

13. Suppose you are given the following list of strings:

```
L = ['aabaabac', 'cabaabca', 'aaabbcba', 'aabacbab', 'acababba']
```

Patterns like this show up in many places, including DNA sequencing. The user has a string of their own with only some letters filled in and the rest as asterisks. An example is `a**a****`. The user would like to know which of the strings in the list fit with their pattern. In the example just given, the matching strings are the first and fourth. One way to solve this problem is to create a dictionary whose keys are the indices in the user's string of the non-asterisk characters and whose values are those characters. Write a program implementing this approach (or some other approach) to find the strings that match a user-entered string.

14. Dictionaries provide a convenient way to store structured data. Here is an example dictionary:

```
d=[{'name':'Todd', 'phone':'555-1414', 'email':'todd@mail.net'},
   {'name':'Helga', 'phone':'555-1618', 'email':'helga@mail.net'},
   {'name':'Princess', 'phone':'555-3141', 'email':''},
   {'name':'LJ', 'phone':'555-2718', 'email':'lj@mail.net'}]
```

Write a program that reads through any dictionary like this and prints the following:

(a) All the users whose phone number ends in an 8

(b) All the users that don't have an email address listed

15. The following problem is from Chapter 6. Try it again, this time using a dictionary whose keys are the names of the time zones and whose values are offsets from the Eastern time zone.

    Write a program that converts a time from one time zone to another. The user enters the time in the usual American way, such as `3:48pm` or `11:26am`. The first time zone the user enters is that of the original time and the second is the desired time zone. The possible time zones are `Eastern`, `Central`, `Mountain`, or `Pacific`.

    ```
    Time: 11:48pm
    Starting zone: Pacific
    Ending zone: Eastern
    2:48am
    ```

16. (a) Write a program that converts Roman numerals into ordinary numbers. Here are the conversions: M=1000, D=500, C=100, L=50, X=10, V=5 I=1. Don't forget about things like IV being 4 and XL being 40.

    (b) Write a program that converts ordinary numbers into Roman numerals