Chapter 14

Dictionaries

lists and tuples are *containers* in which data are collected so that elements of the data can be easily accessed. lists and tuples are also *sequences* in that data are organized in a well defined sequential manner. One element follows another. The first element has an index of 0, the next has an index of 1, and so on. Alternatively, negative indexing can be used to specify an offset from the last element of a list or tuple. We've discussed the use of slices where a portion of a list or tuple is specified by indices that indicate the start and end of the slice. Almost everything we have done in terms of accessing the elements of a list or tuple has been related to the sequential nature of the container and has relied on numeric indexing.

However, as you know from your day-to-day experiences, there are many situations in which we don't think of collections of data in a sequential manner. For example, your name, your height, and your age are all data associated with you, but you don't think about these in any particular order. Your name is simply your name. If we want to store a person's name, height, and age in a Python program, we can easily store this information in a list (if we want the information to be mutable) or a tuple (if we want the information to be immutable). If we do this, the order of an element dictates how it should be interpreted. So, for example, perhaps the first element in a list is a name (a string), the second element is an age (an integer), and the third element is a height (a float or integer in centimeters). Within a program this organization of information can work well, but there are limits to this. What if we want to keep track of many more facts about a person? Assume, perhaps, there are 15 separate pieces of data we want to record and use. The code to manipulate this information can become difficult to understand and maintain if the programmer only has the position within a list as the key to determining how the data should be interpreted.

As an alternative to lists and tuples, Python provides another container known as a dictionary or dict. Dictionaries share some syntactic properties with lists and tuples but there are many important differences. First, dictionaries are *not* sequential collections of data. Instead, dictionaries consist of key-value pairs. To obtain a "value" (i.e., the data of interest), you specify its associated key. In this way you can create a collection of data in which perhaps the keys are the strings 'name', 'age', and 'height'. The order in which Python stores the key-value pairs is not a concern. We merely need to know that when we specify the key 'name', Python will provide the associated name. When we specify the key 'age', Python will provide the associated age. And, when we specify the key 'height', Python will provide the associated height.

From the file: dictionaries.tex

In this chapter we will explore the ways in which dictionaries can be used. You are already familiar with traditional dictionaries in which the "key" is a given word. Using this key, you can look up the definition of the word (i.e., the data associated with this key). This model works well in some ways for visualizing a `dict` in Python. However, in a traditional dictionary all the keys/words are in alphabetical order. There is no such ordering of the keys in Python. Despite this lack of order, one of the important properties of a `dict` is that Python can return the data for a given key extremely quickly. This speed is maintained even when the dictionary is "huge."[1]

14.1 Dictionary Basics

To create a `dict`, we use curly braces, i.e., `{}`. If there is nothing between the braces (other than whitespace), the `dict` is empty. It is not uncommon to start with an empty dictionary and later add key-value pairs. Alternatively, key-value pairs can be specified when the dictionary is created. This is done by separating the key from the value by a colon and separating key-value pairs by a comma. This is illustrated in Listing 14.1 which is discussed below the listing.

Listing 14.1 Creation of dictionaries.

```
1  >>> abe = {'name' : 'Abraham Lincoln', 'age' : 203, 'height' : 193}
2  >>> abe['height']
3  193
4  >>> abe['name']
5  'Abraham Lincoln'
6  >>> james = {}
7  >>> james['name']
8  Traceback (most recent call last):
9    File "<stdin>", line 1, in <module>
10 KeyError: 'name'
11 >>> james['name'] = 'James Madison'
12 >>> james['height'] = 163
13 >>> james['age'] = 261
14 >>> print(abe)
15 {'age': 203, 'name': 'Abraham Lincoln', 'height': 193}
16 >>> james
17 {'age': 261, 'name': 'James Madison', 'height': 163}
```

In line 1 the dictionary `abe` is created with three key-value pairs. The use of whitespace surrounding a colon is optional. Although not done here, the declaration can span multiple lines because the opening brace behaves something like an opening parentheses and the declaration does not end until the corresponding closing brace is entered. To obtain the value associated with a key, the key is specified within square brackets as shown in lines 2 and 4.

[1] Python `dicts` are *hashes*. We will not consider the details of a hash, but essentially what happens in a hash is that a *hash function* is applied to the key. The value this function returns indicates, at least approximately, where the associated value can be found in memory.

An empty `dict` is created in line 6 and assigned to the variable `james`. In line 7 an attempt is made to access the value associated with the key `'name'`. However, since this key doesn't exist in `james`, this produces the `KeyError` exception shown in lines 8 through 10. However, we can add key-value pairs to a dictionary through assignment statements as shown in lines 11 through 13.[2]

In line 14 a `print()` statement is used to display the dictionary `abe`. In the output in line 15, note that the order of key-value pairs is not the same as the order in which we provided them.[3] In the interactive environment, we can see the contents of a dictionary simply by entering the dictionary name and hitting return as is done with `james` in line 16. The output in line 17 again shows that the order in which we specify key-value pairs is unrelated to the order in which Python stores or displays them.

In Listing 14.1 all the keys are strings. But, keys can be numeric values or tuples or, in fact, any immutable object! This is illustrated in Listing 14.2 where, in lines 1 through 4, the dictionary `d` is created with keys that are a string (line 1), an integer (line 2), a `tuple` (line 3), and a `float` (line 4). Lines 5 through 16 demonstrate that the keys do indeed produce the associated values. Note that the value associated with the `tuple` key (line 3) is itself a `dict` and the value associated with the `float` key (in line 4) is a `list`. Although a key cannot be a mutable object, the values associated with a key can be mutable. (`dicts` are mutable, because we can change keys and values.)

Listing 14.2 Demonstration that keys to a `dict` can be any immutable object. The associated values can be either mutable or immutable objects.

```
1  >>> d = {'alma mater' : 'WSU',
2  ...       42 : 'The meaning of life.',
3  ...       (3, 4) : {'first' : 33, 'second' : 3 + 4},
4  ...       5.7 : [5, 0.7]}
5  >>> d['alma mater']
6  'WSU'
7  >>> d[42]
8  'The meaning of life.'
9  >>> d[5.7]
10 [5, 0.7]
11 >>> d[5.7][1]
12 0.7
13 >>> d[(3, 4)]
14 {'second': 7, 'first': 33}
15 >>> d[(3, 4)]['second']
16 7
```

[2] Abraham Lincoln was the tallest President of the United States at 6 feet 4 inches. James Madison was the shortest at 5 feet 4 inches.

[3] The order is dependent on the hash function used. The details of this are something we can easily access and thus we simply need to accept that ordering of values in the dictionary is effectively random. Nevertheless, as we will see, using the `sorted()` function there are ways to order the data from dictionaries.

Let us now take a look at the methods provided by a dictionary. These are shown in Listing 14.3. The methods of interest to us are in slanted bold text. The `keys()` method provides the keys for a `dict` while `values()` provides the values. The method `items()` provides `tuples` of all the key-value pairs. These methods (and the `get()` method) will be discussed further in the following sections.

Listing 14.3 Methods provided by `dicts`.

```
>>> dir({})
['__class__', '__contains__', '__delattr__', '__delitem__', '__doc__',
  '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
  '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__',
  '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
  '__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
  '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'items',
  'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

14.2 Cycling through a Dictionary

Although a `dict` is not a sequence like a `list` or a `tuple`, it is an *iterable* and can be used in a `for`-loop header. This is demonstrated in Listing 14.4 where the dictionary `abe` is created in line 1 and then used as the iterable in the `for`-loop in line 2. Here we use `foo` for the loop variable name to indicate that it is not obvious what value is assigned to this variable. The body of the loop (line 3) simply prints the value of the loop variable. We see in the output in lines 5 through 7 that the loop variable is assigned the values of the keys. By invoking the `keys()` method on the dictionary `abe`, the `for`-loop defined in lines 8 and 9 explicitly says that the iterable should be the keys of the `dict`. This loop is functionally identical to the loop in lines 2 and 3. Since it is superfluous, typically one does not invoke the `keys()` method in the header of a `for`-loop and instead writes the header as shown in line 2. (Another common idiom is to name the loop variable `key`.)

Listing 14.4 When a `dict` is used as the iterable in a `for`-loop, the loop variable takes on the values of keys.

```
>>> abe = {'name' : 'Abraham Lincoln', 'age' : 203, 'height' : 193}
>>> for foo in abe:
...     print(foo)
...
age
name
height
>>> for foo in abe.keys():
...     print(foo)
...
```

```
11 age
12 name
13 height
```

Of course, one is usually interested in the values associated with keys and not in the keys themselves. Listing 14.5 demonstrates the printing of keys together with their associated values. In the `print()` statement in line 3, the second argument is `':\t'`. Recall that `\t` is the escape sequence for a tab. This serves to align the output as shown in lines 5 through 7.

Listing 14.5 Display of key-value pairs.

```
1 >>> abe = {'name' : 'Abraham Lincoln', 'age' : 203, 'height' : 193}
2 >>> for key in abe:
3 ...     print(key, ':\t', abe[key], sep="")   # \t = tab.
4 ...
5 age:    203
6 name:   Abraham Lincoln
7 height: 193
```

An alternative way to display key-value pairs is provided by the `items()` method. As mentioned in the previous section, this method provides a pairing of keys and their associated values. In this way, simultaneous assignment can be used in the header of the `for`-loop to obtain both the key and the associated value. This is demonstrated in Listing 14.6. This listing is identical to Listing 14.5 except in line 2 (where `value` has been added as a loop variable and the `items()` method is invoked) and in line 3 (where `abe[key]` has been replaced by `value`).

Listing 14.6 Use of the `items()` method to obtain both the key and the associated value in the header of the `for`-loop.

```
1 >>> abe = {'name' : 'Abraham Lincoln', 'age' : 203, 'height' : 193}
2 >>> for key, value in abe.items():
3 ...     print(key, ':\t', value, sep="")
4 ...
5 age:    203
6 name:   Abraham Lincoln
7 height: 193
```

If we are interested in obtaining only the values from a `dict`, we can obtain them using the `values()` method. This is demonstrated in Listing 14.7.

Listing 14.7 Displaying the values in a dictionary.

```
1 >>> abe = {'name' : 'Abraham Lincoln', 'age' : 203, 'height' : 193}
2 >>> for value in abe.values():
```

```
...     print(value)
...
203
Abraham Lincoln
193
```

Assume a teacher has created a dictionary of students in which the keys are the students' names. Each student is assigned a grade (which is a string). The teacher then wants to view the students' names and grades. Typically such a listing is presented alphabetically. However, with a `dict` we have no way to directly enforce the ordering of the keys. For a `list` the `sort()` method can be used to order the elements, but this cannot be used with the keys of a `dict` because the keys themselves are not a `list` nor does the `keys()` method produce a `list`. Fortunately, Python provides a function called `sorted()` that can be used to sort the keys. `sorted()` takes an iterable as its argument and returns a `list` of sorted values.

In Listing 14.8, in lines 1 through 5, a dictionary of eight students is created. In lines 6 and 7 a `for`-loop is used to display all the student names and grades. Note that the `sorted()` function is used in the header (line 6) to sort the keys. For strings, `sorted()` will, by default, perform the sort in alphabetical order. The body of the `for`-loop consists of a single `print()` statement. A format string is used to ensure the output appears nicely aligned (because of the plus or minus that may appear in the grade, the width of the first replacement field is set to two characters). Note that the listing of students in lines 9 through 16 is in alphabetical order. The `for`-loop in lines 17 and 18 does not use the `sorted()` function to sort the keys nor is a format string used for the output. The subsequent output, in lines 20 through 27, is not in alphabetical order and the names are no longer aligned.

Listing 14.8 Use of `sorted()` to sort the keys of a `dict` and thus show the data "in order."

```
>>> students = {
...   'Harry' : 'B+',  'Hermione' : 'A+', 'Ron' : 'B-',
...   'Fred' : 'C', 'George': 'C', 'Nevel' : 'B',
...   'Lord Voldemort' : 'F', 'Ginny' : 'A'
... }
>>> for key in sorted(students):    # Sorted keys.
...     print("{:2} {}".format(students[key], key))
...
C  Fred
C  George
A  Ginny
B+ Harry
A+ Hermione
F  Lord Voldemort
B  Nevel
B- Ron
>>> for key in students:            # Unsorted keys.
...     print(students[key], key)
...
```

```
A+ Hermione
B- Ron
B+ Harry
B Nevel
F Lord Voldemort
A Ginny
C George
C Fred
```

14.3 get()

The dictionary method `get()` provides another way to obtain the value associated with a key. However, there are two important differences between the way `get()` behaves and the way dictionaries behave when a key is specified within brackets. The first argument to `get()` is the key. If the key does not exist within the dictionary, no error is produced. Instead, `get()` returns `None`. This is demonstrated in Listing 14.9. In line 1 a dictionary is created with keys `'age'` and `'height'`. The header of the `for`-loop in line 2 explicitly sets the loop variable `key` to `'name'`, `'age'`, and `'height'`. In the body of the loop, in line 3, the `get()` method is used to obtain the value associated with the given key. The output in line 5 shows that the method returns `None` for the key `'name'`. The `for`-loop in lines 8 and 9 is similar to the previous loop except here the values are obtained using `james[key]` rather than `james.get(key)`. This results in an error because the key `'name'` is not defined. This error terminates the loop, i.e., we do not see the other values for which a key is defined.

Listing 14.9 The `get()` method can be used to look up values for a given key. If the key does not exist, the method returns `None`.

```
1  >>> james = {'age': 261, 'height': 163}
2  >>> for key in ['name', 'age', 'height']:
3  ...     print(key, ':\t', james.get(key), sep="")
4  ...
5  name:   None
6  age:    261
7  height: 163
8  >>> for key in ['name', 'age', 'height']:
9  ...     print(key, ':\t', james[key], sep="")
10 ...
11 Traceback (most recent call last):
12   File "<stdin>", line 2, in <module>
13 KeyError: 'name'
```

The other important difference between using `get()` and specifying a key within brackets is that `get()` takes an optional second argument that specifies what should be returned when a key does not exist. In this way we can obtain a value other than `None` for undefined keys. Effectively

this allows us to provide a default value. This is demonstrated in Listing 14.10 which is a slight variation of Listing 14.9. The only difference is in line 3 where the string `John Doe` is provided as the second argument to the `get()` method. Note that this particular default value (i.e., `John Doe`) appears to be reasonable in terms of providing a missing "name," but it does not make much sense as a default for the age or height.

Listing 14.10 Demonstration of the use of a default return value for `get()`.

```
1 >>> james = {'age': 261, 'height': 163}
2 >>> for key in ['name', 'age', 'height']:
3 ...     print(key, ':\t', james.get(key, "John Doe"), sep="")
4 ...
5 name:   John Doe
6 age:    261
7 height: 163
```

Let's now consider a more practical way in which the default value of the `get()` method can be used. Assume we want to analyze some text to determine how often each "word" appears within the text. For the sake of simplicity, we will assume a word is any collection of contiguous non-whitespace characters. Thus, letters, digits, and punctuation marks are all considered part of a word. Furthermore, we will maintain case sensitivity so that words having the same letters but different cases are considered to be different. For example, all of the following are considered to be different words:

```
  end   end.   end,   End   "End
```

Analysis of text in which one obtains the number of occurrences of each word is often referred to as a *concordance*. As an example of this, let's analyze the following text to determine how many times each word appears:

```
How much wood could a woodchuck chuck if a woodchuck could chuck
wood?  A woodchuck you say?  Not much if the wood were mahogany.
```

We can do this rather easily as demonstrated by the code in Listing 14.11. This code is discussed following the listing.

Listing 14.11 Analysis of text to determine the number of times each word appears.

```
1 >>> text = """
2 ... How much wood could a woodchuck chuck if a woodchuck could chuck
3 ... wood?  A woodchuck you say?  Not much if the wood were mahogany.
4 ... """
5 >>> concordance = {}
6 >>> for word in text.split():
7 ...     concordance[word] = concordance.get(word, 0) + 1
8 ...
9 >>> for key in concordance:
```

```
10 ...     print(concordance[key], key)
11 ...
12 2 a
13 2 wood
14 1 A
15 1 mahogany.
16 1 say?
17 2 could
18 2 chuck
19 1 How
20 2 much
21 3 woodchuck
22 1 were
23 1 Not
24 1 you
25 1 wood?
26 1 the
27 2 if
```

In lines 1 through 4 the text is assigned to the variable `text`. In line 5 an empty dictionary is created and assigned to the variable `concordance`. This dictionary will have keys that are the words in the text. The value associated with each key will ultimately be the number of times the word appears in the text.

The `for`-loop in lines 6 and 7 cycles through each word in `text`. In the header, in line 6, this is accomplished by using the `split()` method on `text` to obtain a `list` of all the individual words. The body of the `for`-loop has a single assignment statement (line 7). On the right side of the assignment statement the `get()` method is used to determine the number of previous occurrences of the given key/word. If the word has not been seen before, the `get()` method returns `0` (i.e., the optional second argument is the integer `0`). Otherwise it returns whatever value is already stored in the dictionary. The value that `get()` returns is incremented by one (indicating there has been one more occurrence of the given word) and this is assigned to `concordance` with the given key/word.

The `for`-loop in lines 9 and 10 is simply used to display the concordance, i.e., the count for the number of occurrences of each word. We see, for example, that the word `wood` occurrs twice while `woodchuck` occurrs three times.

14.4 Chapter Summary

Dictionaries consist of key-value pairs. Any object consisting of immutable components can be used as a key. Values may be any object. Dictionaries themselves are mutable.

Dictionaries are unordered collections of data (i.e., they are not sequences).

The value associated with a given key can be obtained by giving the dictionary name followed by the key enclosed in square brackets. For example, for the dictionary `d`, the value associated with the key `k` is given by `d[k]`. It is an error to attempt to access a non-existent key this way.

The **`get()`** method can be used to obtain the value associated with a key. If the key does not exist, by default, `get()` returns `None`. However, an optional argument can be provided to specify the return value for a non-existent key.

The **`keys()`** method returns the keys of a dictionary. When a dictionary is used as an iterable (e.g., in the header of a `for`-loop), by default the iteration is over the keys. Thus, if `d` is a dictionary, "`for k in d:`" is equivalent to "`for k in d.keys()`".

The **`sorted()`** function can be used to sort the keys of a dictionary. Thus, "`for k in sorted(d):`" cycles through the keys in order. (Similar to the `sort()` method for `lists`, the order can be further controlled by providing a key function whose output dictates the values to be used for the sorting. Additionally, setting the optional argument `reverse` to `True` causes `sorted()` to reverse the order of the items.)

The **`values()`** method returns the values in the dictionary. Thus, if `d` is a dictionary, "`for v in d.values():`" cycles through the values of the dictionary.

The **`items()`** method returns the key-value pairs in the dictionary.

14.5 Review Questions

1. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
print(d['c'])
```

   (a) `c`

   (b) `2`

   (c) `'c' : 2`

   (d) This code produces an error.

2. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
print(d[2])
```

   (a) `c`

   (b) `2`

   (c) `'c' : 2`

   (d) This code produces an error.

3. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
print(d.get(2, 'c'))
```

(a) `c`

(b) `2`

(c) `'c' : 2`

(d) This code produces an error.

4. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
for x in sorted(d):
   print(d[x], end=" ")
```

(a) `a b c`

(b) `0 1 2`

(c) `('a', 0) ('b', 1) ('c', 2)`

(d) This code produces an error.

5. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
for x in sorted(d.values()):
   print(x, end=" ")
```

(a) `a b c`

(b) `0 1 2`

(c) `('a', 0) ('b', 1) ('c', 2)`

(d) This code produces an error.

6. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
for x in sorted(d.items()):
   print(x, end=" ")
```

(a) `a b c`

(b) `0 1 2`

(c) `('a', 0) ('b', 1) ('c', 2)`

(d) This code produces an error.

7. What is the output produced by the following code?

```
d = {'a' : 0, 'b': 1, 'c' : 2}
for x in sorted(d.keys()):
   print(x, end=" ")
```

(a) `a b c`

(b) `0 1 2`

(c) `('a', 0) ('b', 1) ('c', 2)`

(d) This code produces an error.

8. What is the output produced by the following code?

```
pres = {'george' : 'washington', 'thomas' : 'jefferson',
     'john' : 'adams'}
print(pres.get('washington', 'dc'))
```

(a) `george`

(b) `washington`

(c) `dc`

(d) This code produces an error.

9. What is the output produced by the following code?

```
pres = {'george' : 'washington', 'thomas' : 'jefferson',
     'john' : 'adams'}
for p in sorted(pres):
    print(p, end=" ")
```

(a) `george thomas john`

(b) `george john thomas`

(c) `washington jefferson adams`

(d) `adams jefferson washington`

(e) None of the above.

ANSWERS: 1) b; 2) d; 3) a; 4) b; 5) b; 6) c; 7) a; 8) c; 9) b;