CHAPTER 9

Dictionaries and Sets

TOPICS

9.1 Dictionaries
9.2 Sets
9.3 Serializing Objects

9.1 Dictionaries

CONCEPT: A dictionary is an object that stores a collection of data. Each element in a dictionary has two parts: a key and a value. You use a key to locate a specific value.

VideoNote Introduction to Dictionaries

When you hear the word "dictionary," you probably think about a large book such as the Merriam-Webster dictionary, containing words and their definitions. If you want to know the meaning of a particular word, you locate it in the dictionary to find its definition.

In Python, a *dictionary* is an object that stores a collection of data. Each element that is stored in a dictionary has two parts: a *key* and a *value*. In fact, dictionary elements are commonly referred to as *key-value pairs*. When you want to retrieve a specific value from a dictionary, you use the key that is associated with that value. This is similar to the process of looking up a word in the Merriam-Webster dictionary, where the words are keys and the definitions are values.

For example, suppose each employee in a company has an ID number, and we want to write a program that lets us look up an employee's name by entering that employee's ID number. We could create a dictionary in which each element contains an employee ID number as the key and that employee's name as the value. If we know an employee's ID number, then we can retrieve that employee's name.

Another example would be a program that lets us enter a person's name and gives us that person's phone number. The program could use a dictionary in which each element contains a person's name as the key and that person's phone number as the value. If we know a person's name, then we can retrieve that person's phone number.

NOTE: Key-value pairs are often referred to as *mappings* because each key is mapped to a value.

Creating a Dictionary

You can create a dictionary by enclosing the elements inside a set of curly braces (`{}`). An element consists of a key, followed by a colon, followed by a value. The elements are separated by commas. The following statement shows an example:

```
phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}
```

This statement creates a dictionary and assigns it to the `phonebook` variable. The dictionary contains the following three elements:

- The first element is `'Chris':'555-1111'`. In this element the key is `'Chris'` and the value is `'555-1111'`.
- The second element is `'Katie':'555-2222'`. In this element the key is `'Katie'` and the value is `'555-2222'`.
- The third element is `'Joanne':'555-3333'`. In this element the key is `'Joanne'` and the value is `'555-3333'`.

In this example the keys and the values are strings. The values in a dictionary can be objects of any type, but the keys must be immutable objects. For example, keys can be strings, integers, floating-point values, or tuples. Keys cannot be lists or any other type of immutable object.

Retrieving a Value from a Dictionary

The elements in a dictionary are not stored in any particular order. For example, look at the following interactive session in which a dictionary is created and its elements are displayed:

```
>>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
'Joanne':'555-3333'} Enter
>>> phonebook Enter
{'Chris': '555-1111', 'Joanne': '555-3333', 'Katie': '555-2222'}
>>>
```

Notice that the order in which the elements are displayed is different than the order in which they were created. This illustrates how dictionaries are not sequences, like lists, tuples, and strings. As a result, you cannot use a numeric index to retrieve a value by its position from a dictionary. Instead, you use a key to retrieve a value.

To retrieve a value from a dictionary, you simply write an expression in the following general format:

```
dictionary_name[key]
```

In the general format, *`dictionary_name`* is the variable that references the dictionary, and *`key`* is a key. If the key exists in the dictionary, the expression returns the value that is associated with the key. If the key does not exist, a `KeyError` exception is raised. The following interactive session demonstrates:

```
1   >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
    'Joanne':'555-3333'} Enter
2   >>> phonebook['Chris'] Enter
3   '555-1111'
4   >>> phonebook['Joanne'] Enter
5   '555-3333'
6   >>> phonebook['Katie'] Enter
7   '555-2222'
8   >>> phonebook['Kathryn'] Enter
Traceback (most recent call last):
    File "<pyshell#5>", line 1, in <module>
      phonebook['Kathryn']
KeyError: 'Kathryn'
>>>
```

Let's take a closer look at the session:

- Line 1 creates a dictionary containing names (as keys) and phone numbers (as values).
- In line 2, the expression `phonebook['Chris']` returns the value from the `phonebook` dictionary that is associated with the key `'Chris'`. The value is displayed in line 3.
- In line 4, the expression `phonebook['Joanne']` returns the value from the `phonebook` dictionary that is associated with the key `'Joanne'`. The value is displayed in line 5.
- In line 6, the expression `phonebook['Katie']` returns the value from the `phonebook` dictionary that is associated with the key `'Katie'`. The value is displayed in line 7.
- In line 8, the expression `phonebook['Kathryn']` is entered. There is no such key as `'Kathryn'` in the `phonebook` dictionary, so a `KeyError` exception is raised.

NOTE: Remember that string comparisons are case sensitive. The expression `phonebook['katie']` will not locate the key `'Katie'` in the dictionary.

Using the `in` and `not in` Operators to Test for a Value in a Dictionary

As previously demonstrated, a `KeyError` exception is raised if you try to retrieve a value from a dictionary using a nonexistent key. To prevent such an exception, you can use the `in` operator to determine whether a key exists before you try to use it to retrieve a value. The following interactive session demonstrates:

```
1   >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
    'Joanne':'555-3333'} Enter
2   >>> if 'Chris' in phonebook: Enter
3           print(phonebook['Chris']) Enter Enter
4
5   555-1111
6   >>>
```

The `if` statement in line 2 determines whether the key `'Chris'` is in the `phonebook` dictionary. If it is, the statement in line 3 displays the value that is associated with that key.

You can also use the `not in` operator to determine whether a key does not exist, as demonstrated in the following session:

```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222'} Enter
2  >>> if 'Joanne' not in phonebook: Enter
3          print('Joanne is not found.') Enter Enter
4
5  Joanne is not found.
6  >>>
```

NOTE: Keep in mind that string comparisons with the `in` and `not in` operators are case sensitive.

Adding Elements to an Existing Dictionary

Dictionaries are mutable objects. You can add new key-value pairs to a dictionary with an assignment statement in the following general format:

```
dictionary_name[key] = value
```

In the general format, *`dictionary_name`* is the variable that references the dictionary, and *`key`* is a key. If *`key`* already exists in the dictionary, its associated value will be changed to *`value`*. If the *`key`* does not exist, it will be added to the dictionary, along with *`value`* as its associated value. The following interactive session demonstrates:

```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
   'Joanne':'555-3333'} Enter
2  >>> phonebook['Joe'] = '555-0123' Enter
3  >>> phonebook['Chris'] = '555-4444' Enter
4  >>> phonebook Enter
5  {'Chris': '555-4444', 'Joanne': '555-3333', 'Joe': '555-0123',
   'Katie': '555-2222'}
6  >>>
```

Let's take a closer look at the session:

- Line 1 creates a dictionary containing names (as keys) and phone numbers (as values).
- The statement in line 2 adds a new key-value pair to the `phonebook` dictionary. Because there is no key `'Joe'` in the dictionary, this statement adds the key `'Joe'`, along with its associated value `'555-0123'`.
- The statement in line 3 changes the value that is associated with an existing key. Because the key `'Chris'` already exists in the `phonebook` dictionary, this statement changes its associated value to `'555-4444'`.
- Line 4 displays the contents of the `phonebook` dictionary. The output is shown in line 5.

NOTE: You cannot have duplicate keys in a dictionary. When you assign a value to an existing key, the new value replaces the existing value.

Deleting Elements

You can delete an existing key-value pair from a dictionary with the `del` statement. Here is the general format:

```
del dictionary_name[key]
```

In the general format, *`dictionary_name`* is the variable that references the dictionary, and *`key`* is a key. After the statement executes, the *`key`* and its associated value will be deleted from the dictionary. If the *`key`* does not exist, a `KeyError` exception is raised. The following interactive session demonstrates:

```
 1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
    'Joanne':'555-3333'} Enter
 2  >>> phonebook Enter
 3  {'Chris': '555-1111', 'Joanne': '555-3333', 'Katie': '555-2222'}
 4  >>> del phonebook['Chris'] Enter
 5  >>> phonebook Enter
 6  {'Joanne': '555-3333', 'Katie': '555-2222'}
 7  >>> del phonebook['Chris'] Enter
 8  Traceback (most recent call last):
 9      File "<pyshell#5>", line 1, in <module>
10          del phonebook['Chris']
11  KeyError: 'Chris'
12  >>>
```

Let's take a closer look at the session:

- Line 1 creates a dictionary, and line 2 displays its contents.
- Line 4 deletes the element with the key `'Chris'`, and line 5 displays the contents of the dictionary. You can see in the output in line 6 that the element no longer exists in the dictionary.
- Line 7 tries to delete the element with the key `'Chris'` again. Because the element no longer exists, a `KeyError` exception is raised.

To prevent a `KeyError` exception from being raised, you should use the `in` operator to determine whether a key exists before you try to delete it and its associated value. The following interactive session demonstrates:

```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222',
   'Joanne':'555-3333'} Enter
2  >>> if 'Chris' in phonebook: Enter
3          del phonebook['Chris'] Enter Enter
4
5  >>> phonebook Enter
6  {'Joanne': '555-3333', 'Katie': '555-2222'}
7  >>>
```

Getting the Number of Elements in a Dictionary

You can use the built-in `len` function to get the number of elements in a dictionary. The following interactive session demonstrates:

```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222'} Enter
2  >>> num_items = len(phonebook) Enter
```

```
3   >>> print(num_items) Enter
4   2
5   >>>
```

Here is a summary of the statements in the session:

- Line 1 creates a dictionary with two elements and assigns it to the `phonebook` variable.
- Line 2 calls the `len` function passing the `phonebook` variable as an argument. The function returns the value 2, which is assigned to the `num_items` variable.
- Line 3 passes `num_items` to the `print` function. The function's output is shown in line 4.

Mixing Data Types in a Dictionary

As previously mentioned, the keys in a dictionary must be immutable objects, but their associated values can be any type of object. For example, the values can be lists, as demonstrated in the following interactive session. In this session we create a dictionary in which the keys are student names and the values are lists of test scores.

```
1   >>> test_scores = { 'Kayla' : [88, 92, 100], Enter
2                       'Luis' : [95, 74, 81], Enter
3                       'Sophie' : [72, 88, 91], Enter
4                       'Ethan' : [70, 75, 78] } Enter
5   >>> test_scores Enter
6   {'Kayla': [88, 92, 100], 'Sophie': [72, 88, 91], 'Ethan': [70,
    75, 78],
7   'Luis': [95, 74, 81]}
8   >>> test_scores['Sophie'] Enter
9   [72, 88, 91]
10  >>> kayla_scores = test_scores['Kayla'] Enter
11  >>> print(kayla_scores) Enter
12  [88, 92, 100]
13  >>>
```

Let's take a closer look at the session. This statement in lines 1 through 4 creates a dictionary and assigns it to the `test_scores` variable. The dictionary contains the following four elements:

- The first element is `'Kayla' : [88, 92, 100]`. In this element the key is `'Kayla'` and the value is the list `[88, 92, 100]`.
- The second element is `'Luis' : [95, 74, 81]`. In this element the key is `'Luis'` and the value is the list `[95, 74, 81]`.
- The third element is `'Sophie' : [72, 88, 91]`. In this element the key is `'Sophie'` and the value is the list `[72, 88, 91]`.
- The fourth element is `'Ethan' : [70, 75, 78]`. In this element the key is `'Ethan'` and the value is the list `[70, 75, 78]`.

Here is a summary of the rest of the session:

- Line 5 displays the contents of the dictionary, as shown in lines 6 through 7.
- Line 8 retrieves the value that is associated with the key `'Sophie'`. The value is displayed in line 9.

- Line 10 retrieves the value that is associated with the key `'Kayla'` and assigns it to the `kayla_scores` variable. After this statement executes, the `kayla_scores` variable references the list `[88, 92, 100]`.
- Line 11 passes the `kayla_scores` variable to the `print` function. The function's output is shown in line 12.

The values that are stored in a single dictionary can be of different types. For example, one element's value might be a string, another element's value might be a list, and yet another element's value might be an integer. The keys can be of different types, too, as long as they are immutable. The following interactive session demonstrates how different types can be mixed in a dictionary:

```
1  >>> mixed_up = {'abc':1, 999:'yada yada', (3, 6, 9):[3, 6, 9]} Enter
2  >>> mixed_up Enter
3  {(3, 6, 9): [3, 6, 9], 'abc': 1, 999: 'yada yada'}
4  >>>
```

This statement in line 1 creates a dictionary and assigns it to the `mixed_up` variable. The dictionary contains the following elements:

- The first element is `'abc':1`. In this element the key is the string `'abc'` and the value is the integer 1.
- The second element is `999:'yada yada'`. In this element the key is the integer 999 and the value is the string `'yada yada'`.
- The third element is `(3, 6, 9):[3, 6, 9]`. In this element the key is the tuple `(3, 6, 9)` and the value is the list `[3, 6, 9]`.

The following interactive session gives a more practical example. It creates a dictionary that contains various pieces of data about an employee:

```
1  >>> employee = {'name' : 'Kevin Smith', 'id' : 12345, 'payrate' :
   25.75 } Enter
2  >>> employee Enter
3  {'payrate': 25.75, 'name': 'Kevin Smith', 'id': 12345}
4  >>>
```

This statement in line 1 creates a dictionary and assigns it to the `employee` variable. The dictionary contains the following elements:

- The first element is `'name' : 'Kevin Smith'`. In this element the key is the string `'name'` and the value is the string `'Kevin Smith'`.
- The second element is `'id' : 12345`. In this element the key is the string `'id'` and the value is the integer 12345.
- The third element is `'payrate' : 25.75`. In this element the key is the string `'payrate'` and the value is the floating-point number 25.75.

Creating an Empty Dictionary

Sometimes you need to create an empty dictionary and then add elements to it as the program executes. You can use an empty set of curly braces to create an empty dictionary, as demonstrated in the following interactive session:

```
1  >>> phonebook = {} Enter
2  >>> phonebook['Chris'] = '555-1111' Enter
```

```
3   >>> phonebook['Katie'] = '555-2222' Enter
4   >>> phonebook['Joanne'] = '555-3333' Enter
5   >>> phonebook Enter
6   {'Chris': '555-1111', 'Joanne': '555-3333', 'Katie': '555-2222'}
7   >>>
```

The statement in line 1 creates an empty dictionary and assigns it to the `phonebook` variable. Lines 2 through 4 add key-value pairs to the dictionary, and the statement in line 5 displays the dictionary's contents.

You can also use the built-in `dict()` method to create an empty dictionary, as shown in the following statement:

```
phonebook = dict()
```

After this statement executes, the `phonebook` variable will reference an empty dictionary.

Using the `for` Loop to Iterate over a Dictionary

You can use the `for` loop in the following general format to iterate over all the keys in a dictionary:

```
for var in dictionary:
    statement
    statement
    etc.
```

In the general format, *`var`* is the name of a variable and *`dictionary`* is the name of a dictionary. This loop iterates once for each element in the dictionary. Each time the loop iterates, *`var`* is assigned a key. The following interactive session demonstrates:

```
 1  >>> phonebook = {'Chris':'555-1111', Enter
 2                  'Katie':'555-2222', Enter
 3                  'Joanne':'555-3333'} Enter
 4  >>> for key in phonebook: Enter
 5          print(key) Enter Enter
 6
 7
 8  Chris
 9  Joanne
10  Katie
11  >>> for key in phonebook: Enter
12          print(key, phonebook[key]) Enter Enter
13
14
15  Chris 555-1111
16  Joanne 555-3333
17  Katie 555-2222
18  >>>
```

Here is a summary of the statements in the session:

- Lines 1 through 3 create a dictionary with three elements and assign it to the `phonebook` variable.

- Lines 4 through 5 contain a `for` loop that iterates once for each element of the `phonebook` dictionary. Each time the loop iterates, the `key` variable is assigned a key. Line 5 prints the value of the `key` variable. Lines 8 through 9 show the output of the loop.
- Lines 11 through 12 contain another `for` loop that iterates once for each element of the `phonebook` dictionary, assigning a key to the `key` variable. Line 5 prints the `key` variable, followed by the value that is associated with that key. Lines 15 through 17 show the output of the loop.

Some Dictionary Methods

Dictionary objects have several methods. In this section we look at some of the more useful ones, which are summarized in Table 9-1.

Table 9-1 Some of the dictionary methods

Method	Description
`clear`	Clears the contents of a dictionary.
`get`	Gets the value associated with a specified key. If the key is not found, the method does not raise an exception. Instead, it returns a default value.
`items`	Returns all the keys in a dictionary and their associated values as a sequence of tuples.
`keys`	Returns all the keys in a dictionary as a sequence of tuples.
`pop`	Returns the value associated with a specified key and removes that key-value pair from the dictionary. If the key is not found, the method returns a default value.
`popitem`	Returns a randomly selected key-value pair as a tuple from the dictionary and removes that key-value pair from the dictionary.
`values`	Returns all the values in the dictionary as a sequence of tuples.

The `clear` Method

The `clear` method deletes all the elements in a dictionary, leaving the dictionary empty. The method's general format is

```
dictionary.clear()
```

The following interactive session demonstrates the method:

```
>>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222'} Enter
>>> phonebook Enter
{'Chris': '555-1111', 'Katie': '555-2222'}
>>> phonebook.clear() Enter
>>> phonebook Enter
{}
>>>
```

Notice that after the statement in line 4 executes, the `phonebook` dictionary contains no elements.

The `get` Method

You can use the `get` method as an alternative to the `[]` operator for getting a value from a dictionary. The `get` method does not raise an exception if the specified key is not found. Here is the method's general format:

```
dictionary.get(key, default)
```

In the general format, *`dictionary`* is the name of a dictionary, *`key`* is a key to search for in the dictionary, and *`default`* is a default value to return if the *`key`* is not found. When the method is called, it returns the value that is associated with the specified *`key`*. If the specified *`key`* is not found in the dictionary, the method returns *`default`*. The following interactive session demonstrates:

```
1  >>> phonebook = {'Chris':'555-1111', 'Katie':'555-2222'} Enter
2  >>> value = phonebook.get('Katie', 'Entry not found') Enter
3  >>> print(value) Enter
4  555-2222
5  >>> value = phonebook.get('Andy', 'Entry not found') Enter
6  >>> print(value) Enter
7  Entry not found
8  >>>
```

Let's take a closer look at the session:

- The statement in line 2 searches for the key `'Katie'` in the `phonebook` dictionary. The key is found, so its associated value is returned and assigned to the `value` variable.
- Line 3 passes the `value` variable to the `print` function. The function's output is shown in line 4.
- The statement in line 5 searches for the key `'Andy'` in the `phonebook` dictionary. The key is not found, so the string `'Entry not found'` is assigned to the `value` variable.
- Line 6 passes the `value` variable to the `print` function. The function's output is shown in line 7.

The `items` Method

The `items` method returns all of a dictionary's keys and their associated values. They are returned as a special type of sequence known as a *dictionary view*. Each element in the dictionary view is a tuple, and each tuple contains a key and its associated value. For example, suppose we have created the following dictionary:

```
phonebook={'Chris':'555-1111','Katie':'555-2222','Joanne':'555-3333'}
```

If we call the `phonebook.items()` method, it returns the following sequence:

```
[('Chris', '555-1111'), ('Joanne', '555-3333'), ('Katie', '555-2222')]
```

Notice the following:

- The first element in the sequence is the tuple `('Chris', '555-1111')`.
- The second element in the sequence is the tuple `('Joanne', '555-3333')`.
- The third element in the sequence is the tuple `('Katie', '555-2222')`.

You can use the `for` loop to iterate over the tuples in the sequence. The following interactive session demonstrates:

```
 1  >>> phonebook = {'Chris':'555-1111', Enter
 2                  'Katie':'555-2222', Enter
 3                  'Joanne':'555-3333'} Enter
 4  >>> for key, value in phonebook.items(): Enter
 5          print(key, value) Enter Enter
 6
 7
 8  Chris 555-1111
 9  Joanne 555-3333
10  Katie 555-2222
11  >>>
```

Here is a summary of the statements in the session:

- Lines 1 through 3 create a dictionary with three elements and assign it to the `phonebook` variable.
- The `for` loop in lines 4 through 5 calls the `phonebook.items()` method, which returns a sequence of tuples containing the key-value pairs in the dictionary. The loop iterates once for each tuple in the sequence. Each time the loop iterates, the values of a tuple are assigned to the `key` and `value` variables. Line 5 prints the value of the `key` variable, followed by the value of the `value` variable. Lines 8 through 10 show the output of the loop.

The `keys` Method

The `keys` method returns all of a dictionary's keys as a dictionary view, which is a type of sequence. Each element in the dictionary view is a key from the dictionary. For example, suppose we have created the following dictionary:

```
phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}
```

If we call the `phonebook.keys()` method, it will return the following sequence:

```
['Chris', 'Joanne', 'Katie']
```

The following interactive session shows how you can use a `for` loop to iterate over the sequence that is returned from the `keys` method:

```
 1  >>> phonebook = {'Chris':'555-1111', Enter
 2                  'Katie':'555-2222', Enter
 3                  'Joanne':'555-3333'} Enter
 4  >>> for key in phonebook.keys(): Enter
 5          print(key) Enter Enter
 6
 7
 8  Chris
 9  Joanne
10  Katie
11  >>>
```

The pop Method

The pop method returns the value associated with a specified key and removes that key-value pair from the dictionary. If the key is not found, the method returns a default value. Here is the method's general format:

```
dictionary.pop(key, default)
```

In the general format, *dictionary* is the name of a dictionary, *key* is a key to search for in the dictionary, and *default* is a default value to return if the *key* is not found. When the method is called, it returns the value that is associated with the specified *key*, and it removes that key-value pair from the dictionary. If the specified *key* is not found in the dictionary, the method returns *default*. The following interactive session demonstrates:

```
 1  >>> phonebook = {'Chris':'555-1111', Enter
 2                   'Katie':'555-2222', Enter
 3                   'Joanne':'555-3333'} Enter
 4  >>> phone_num = phonebook.pop('Chris', 'Entry not found') Enter
 5  >>> phone_num Enter
 6  '555-1111'
 7  >>> phonebook Enter
 8  {'Joanne': '555-3333', 'Katie': '555-2222'}
 9  >>> phone_num = phonebook.pop('Andy', 'Element not found') Enter
10  >>> phone_num Enter
11  'Element not found'
12  >>> phonebook Enter
13  {'Joanne': '555-3333', 'Katie': '555-2222'}
14  >>>
```

Here is a summary of the statements in the session:

- Lines 1 through 3 create a dictionary with three elements and assign it to the phonebook variable.
- Line 4 calls the `phonebook.pop()` method, passing `'Chris'` as the key to search for. The value that is associated with the key `'Chris'` is returned and assigned to the `phone_num` variable. The key-value pair containing the key `'Chris'` is removed from the dictionary.
- Line 5 displays the value assigned to the `phone_num` variable. The output is displayed in line 6. Notice that this is the value that was associated with the key `'Chris'`.
- Line 7 displays the contents of the `phonebook` dictionary. The output is shown in line 8. Notice the key-value pair that contained the key `'Chris'` is no longer in the dictionary.
- Line 9 calls the `phonebook.pop()` method, passing `'Andy'` as the key to search for. The key is not found, so the string `'Entry not found'` is assigned to the `phone_num` variable.
- Line 10 displays the value assigned to the `phone_num` variable. The output is displayed in line 11.
- Line 12 displays the contents of the `phonebook` dictionary. The output is shown in line 13.

The `popitem` Method

The `popitem` method returns a randomly selected key-value pair, and it removes that key-value pair from the dictionary. The key-value pair is returned as a tuple. Here is the method's general format:

```
dictionary.popitem()
```

You can use an assignment statement in the following general format to assign the returned key and value to individual variables:

```
k, v = dictionary.popitem()
```

This type of assignment is known as a *multiple assignment* because multiple variables are being assigned at once. In the general format, *`k`* and *`v`* are variables. After the statement executes, *`k`* is assigned a randomly selected key from the *`dictionary`*, and *`v`* is assigned the value associated with that key. The key-value pair is removed from the *`dictionary`*.

The following interactive session demonstrates:

```
 1  >>> phonebook = {'Chris':'555-1111', Enter
 2                   'Katie':'555-2222', Enter
 3                   'Joanne':'555-3333'} Enter
 4  >>> phonebook Enter
 5  {'Chris': '555-1111', 'Joanne': '555-3333', 'Katie': '555-2222'}
 6  >>> key, value = phonebook.popitem() Enter
 7  >>> print(key, value) Enter
 8  Chris 555-1111
 9  >>> phonebook Enter
10  {'Joanne': '555-3333', 'Katie': '555-2222'}
11  >>>
```

Here is a summary of the statements in the session:

- Lines 1 through 3 create a dictionary with three elements and assign it to the `phonebook` variable.
- Line 4 displays the dictionary's contents, shown in line 5.
- Line 6 calls the `phonebook.popitem()` method. The key and value that are returned from the method are assigned to the variables `key` and `value`. The key-value pair is removed from the dictionary.
- Line 7 displays the values assigned to the `key` and `value` variables. The output is shown in line 8.
- Line 9 displays the contents of the dictionary. The output is shown in line 10. Notice that the key-value pair that was returned from the `popitem` method in line 6 has been removed.

Keep in mind that the `popitem` method raises a `KeyError` exception if it is called on an empty dictionary.

The `values` Method

The `values` method returns all a dictionary's values (without their keys) as a dictionary view, which is a type of sequence. Each element in the dictionary view is a value from the dictionary. For example, suppose we have created the following dictionary:

```
phonebook = {'Chris':'555-1111', 'Katie':'555-2222', 'Joanne':'555-3333'}
```

If we call the `phonebook.values()` method, it returns the following sequence:

```
['555-1111', '555-2222', '555-3333']
```

The following interactive session shows how you can use a `for` loop to iterate over the sequence that is returned from the `values` method:

```
>>> phonebook = {'Chris':'555-1111', Enter
              'Katie':'555-2222', Enter
              'Joanne':'555-3333'} Enter
>>> for val in phonebook.values(): Enter
        print(val) Enter Enter


555-1111
555-3333
555-2222
>>>
```

In the Spotlight:

Using a Dictionary to Simulate a Deck of Cards

In some games involving poker cards, the cards are assigned numeric values. For example, in the game of Blackjack, the cards are given the following numeric values:

- Numeric cards are assigned the value they have printed on them. For example, the value of the 2 of spades is 2, and the value of the 5 of diamonds is 5.
- Jacks, queens, and kings are valued at 10.
- Aces are valued at either 1 or 11, depending on the player's choice.

In this section we look at a program that uses a dictionary to simulate a standard deck of poker cards, where the cards are assigned numeric values similar to those used in Blackjack. (In the program, we assign the value 1 to all aces.) The key-value pairs use the name of the card as the key and the card's numeric value as the value. For example, the key-value pair for the queen of hearts is

```
'Queen of Hearts':10
```

And the key-value pair for the 8 of diamonds is

```
'8 of Diamonds':8
```

The program prompts the user for the number of cards to deal, and it randomly deals a hand of that many cards from the deck. The names of the cards are displayed, as well as the total numeric value of the hand. Program 9-1 shows the program code. The program is divided into three functions: `main`, `create_deck`, and `deal_cards`. Rather than presenting the entire program at once, let's first examine the `main` function:

Program 9-1 (card_dealer.py: `main` function)

```
 1  # This program uses a dictionary as a deck of cards.
 2
 3  def main():
 4      # Create a deck of cards.
 5      deck = create_deck()
 6
 7      # Get the number of cards to deal.
 8      num_cards = int(input('How many cards should I deal? '))
 9
10      # Deal the cards.
11      deal_cards(deck, num_cards)
12
```

Line 5 calls the `create_deck` function. The function creates a dictionary containing the key-value pairs for a deck of cards, and it returns a reference to the dictionary. The reference is assigned to the `deck` variable.

Line 8 prompts the user to enter the number of cards to deal. The input is converted to an `int` and assigned to the `num_cards` variable.

Line 11 calls the `deal_cards` function passing the `deck` and `num_cards` variables as arguments. The `deal_cards` function deals the specified number of cards from the deck.

Next is the `create_deck` function.

Program 9-1 (card_dealer.py: `create_deck` function)

```
13  # The create_deck function returns a dictionary
14  # representing a deck of cards.
15  def create_deck():
16      # Create a dictionary with each card and its value
17      # stored as key-value pairs.
18      deck = {'Ace of Spades':1, '2 of Spades':2, '3 of Spades':3,
19              '4 of Spades':4, '5 of Spades':5, '6 of Spades':6,
20              '7 of Spades':7, '8 of Spades':8, '9 of Spades':9,
21              '10 of Spades':10, 'Jack of Spades':10,
22              'Queen of Spades':10, 'King of Spades': 10,
23
24              'Ace of Hearts':1, '2 of Hearts':2, '3 of Hearts':3,
25              '4 of Hearts':4, '5 of Hearts':5, '6 of Hearts':6,
26              '7 of Hearts':7, '8 of Hearts':8, '9 of Hearts':9,
27              '10 of Hearts':10, 'Jack of Hearts':10,
28              'Queen of Hearts':10, 'King of Hearts': 10,
29
30              'Ace of Clubs':1, '2 of Clubs':2, '3 of Clubs':3,
31              '4 of Clubs':4, '5 of Clubs':5, '6 of Clubs':6,
```

(program continues)

Program 9-1 *(continued)*

```
32                '7 of Clubs':7, '8 of Clubs':8, '9 of Clubs':9,
33                '10 of Clubs':10, 'Jack of Clubs':10,
34                'Queen of Clubs':10, 'King of Clubs': 10,
35
36                'Ace of Diamonds':1, '2 of Diamonds':2, '3 of Diamonds':3,
37                '4 of Diamonds':4, '5 of Diamonds':5, '6 of Diamonds':6,
38                '7 of Diamonds':7, '8 of Diamonds':8, '9 of Diamonds':9,
39                '10 of Diamonds':10, 'Jack of Diamonds':10,
40                'Queen of Diamonds':10, 'King of Diamonds': 10}
41
42        # Return the deck.
43        return deck
44
```

The code in lines 18 through 40 creates a dictionary with key-value pairs representing the cards in a standard poker deck. (The blank lines that appear in lines 22, 29, and 35 were inserted to make the code easier to read.)

Line 43 returns a reference to the dictionary.

Next is the `deal_cards` function.

Program 9-1 (card_dealer.py: `deal_cards` function)

```
45    # The deal_cards function deals a specified number of cards
46    # from the deck.
47
48    def deal_cards(deck, number):
49        # Initialize an accumulator for the hand value.
50        hand_value = 0
51
52        # Make sure the number of cards to deal is not
53        # greater than the number of cards in the deck.
54        if number > len(deck):
55            number = len(deck)
56
57        # Deal the cards and accumulate their values.
58        for count in range(number):
59            card, value = deck.popitem()
60            print(card)
61            hand_value += value
62
63        # Display the value of the hand.
64        print('Value of this hand:', hand_value)
65
66    # Call the main function.
67    main()
```

The `deal_cards` function accepts two arguments: the number of cards to deal and the deck to deal them from. Line 50 initializes an accumulator variable named `hand_value` to 0. The `if` statement in line 54 determines whether the number of cards to deal is greater than the number of cards in the deck. If so, line 55 sets the number of cards to deal to the number of cards in the deck.

The `for` loop that begins in line 58 iterates once for each card that is to be dealt. Inside the loop, the statement in line 59 calls the `popitem` method to randomly return a key-value pair from the `deck` dictionary. The key is assigned to the `card` variable, and the value is assigned to the `value` variable. Line 60 displays the name of the card, and line 61 adds the card's value to the `hand_value` accumulator.

After the loop has finished, line 64 displays the total value of the hand.

Program Output (with input shown in bold)

```
How many cards should I deal? 5 Enter
8 of Hearts
5 of Diamonds
5 of Hearts
Queen of Clubs
10 of Spades
Value of this hand: 38
```

In the Spotlight:

Storing Names and Birthdays in a Dictionary

In this section we look at a program that keeps your friends' names and birthdays in a dictionary. Each entry in the dictionary uses a friend's name as the key and that friend's birthday as the value. You can use the program to look up your friends' birthdays by entering their names.

The program displays a menu that allows the user to make one of the following choices:

1. Look up a birthday
2. Add a new birthday
3. Change a birthday
4. Delete a birthday
5. Quit the program

The program initially starts with an empty dictionary, so you have to choose item 2 from the menu to add a new entry. Once you have added a few entries, you can choose item 1 to look up a specific person's birthday, item 3 to change an existing birthday in the dictionary, item 4 to delete a birthday from the dictionary, or item 5 to quit the program.

Program 9-2 shows the program code. The program is divided into six functions: `main`, `get_menu_choice`, `look_up`, `add`, `change`, and `delete`. Rather than presenting the entire program at once, let's first examine the global constants and the `main` function:

Program 9-2 (birthdays.py: `main` function)

```
 1  # This program uses a dictionary to keep friends'
 2  # names and birthdays.
 3
 4  # Global constants for menu choices
 5  LOOK_UP = 1
 6  ADD = 2
 7  CHANGE = 3
 8  DELETE = 4
 9  QUIT = 5
10
11  # main function
12  def main():
13      # Create an empty dictionary.
14      birthdays = {}
15
16      # Initialize a variable for the user's choice.
17      choice = 0
18
19      while choice != QUIT:
20          # Get the user's menu choice.
21          choice = get_menu_choice()
22
23          # Process the choice.
24          if choice == LOOK_UP:
25              look_up(birthdays)
26          elif choice == ADD:
27              add(birthdays)
28          elif choice == CHANGE:
29              change(birthdays)
30          elif choice == DELETE:
31              delete(birthdays)
32
```

The global constants that are declared in lines 5 through 9 are used to test the user's menu selection. Inside the `main` function, line 14 creates an empty dictionary referenced by the `birthdays` variable. Line 17 initializes the `choice` variable with the value 0. This variable holds the user's menu selection.

The `while` loop that begins in line 19 repeats until the user chooses to quit the program. Inside the loop, line 21 calls the `get_menu_choice` function. The `get_menu_choice` function displays the menu and returns the user's selection. The value that is returned is assigned to the `choice` variable.

The `if-elif` statement in lines 24 through 31 processes the user's menu choice. If the user selects item 1, line 25 calls the `look_up` function. If the user selects item 2, line 27 calls the `add` function. If the user selects item 3, line 29 calls the `change` function. If the user selects item 4, line 31 calls the `delete` function.

The `get_menu_choice` function is next.

Program 9-2 (birthdays.py: `get_menu_choice` function)

```
33  # The get_menu_choice function displays the menu
34  # and gets a validated choice from the user.
35  def get_menu_choice():
36      print()
37      print('Friends and Their Birthdays')
38      print('---------------------------')
39      print('1. Look up a birthday')
40      print('2. Add a new birthday')
41      print('3. Change a birthday')
42      print('4. Delete a birthday')
43      print('5. Quit the program')
44      print()
45
46      # Get the user's choice.
47      choice = int(input('Enter your choice: '))
48
49      # Validate the choice.
50      while choice < LOOK_UP or choice > QUIT:
51          choice = int(input('Enter a valid choice: '))
52
53      # return the user's choice.
54      return choice
55
```

The statements in lines 36 through 44 display the menu on the screen. Line 47 prompts the user to enter his or her choice. The input is converted to an `int` and assigned to the `choice` variable. The `while` loop in lines 50 through 51 validates the user's input and, if necessary, prompts the user to reenter his or her choice. Once a valid choice is entered, it is returned from the function in line 54.

The `look_up` function is next.

Program 9-2 (birthdays.py: `look_up` function)

```
56  # The look_up function looks up a name in the
57  # birthdays dictionary.
58  def look_up(birthdays):
59      # Get a name to look up.
```

(program continues)

Program 9-2 *(continued)*

```
60         name = input('Enter a name: ')
61
62         # Look it up in the dictionary.
63         print(birthdays.get(name, 'Not found.'))
64
```

The purpose of the `look_up` function is to allow the user to look up a friend's birthday. It accepts the dictionary as an argument. Line 60 prompts the user to enter a name, and line 63 passes that name as an argument to the dictionary's `get` function. If the name is found, its associated value (the friend's birthday) is returned and displayed. If the name is not found, the string '`Not found.`' is displayed.

The `add` function is next.

Program 9-2 (birthdays.py: `add` function)

```
65     # The add function adds a new entry into the
66     # birthdays dictionary.
67     def add(birthdays):
68         # Get a name and birthday.
69         name = input('Enter a name: ')
70         bday = input('Enter a birthday: ')
71
72         # If the name does not exist, add it.
73         if name not in birthdays:
74             birthdays[name] = bday
75         else:
76             print('That entry already exists.')
77
```

The purpose of the `add` function is to allow the user to add a new birthday to the dictionary. It accepts the dictionary as an argument. Lines 69 and 70 prompt the user to enter a name and a birthday. The `if` statement in line 73 determines whether the name is not already in the dictionary. If not, line 74 adds the new name and birthday to the dictionary. Otherwise, a message indicating that the entry already exists is printed in line 76.

The `change` function is next.

Program 9-2 (birthdays.py: `change` function)

```
78     # The change function changes an existing
79     # entry in the birthdays dictionary.
80     def change(birthdays):
81         # Get a name to look up.
82         name = input('Enter a name: ')
83
```

```
 84         if name in birthdays:
 85             # Get a new birthday.
 86             bday = input('Enter the new birthday: ')
 87 
 88             # Update the entry.
 89             birthdays[name] = bday
 90         else:
 91             print('That name is not found.')
 92 
```

The purpose of the `change` function is to allow the user to change an existing birthday in the dictionary. It accepts the dictionary as an argument. Line 82 gets a name from the user. The `if` statement in line 84 determines whether the name is in the dictionary. If so, line 86 gets the new birthday, and line 89 stores that birthday in the dictionary. If the name is not in the dictionary, line 91 prints a message indicating so.

The `delete` function is next.

Program 9-2 (birthdays.py: `change` function)

```
 93  # The delete function deletes an entry from the
 94  # birthdays dictionary.
 95  def delete(birthdays):
 96      # Get a name to look up.
 97      name = input('Enter a name: ')
 98 
 99      # If the name is found, delete the entry.
100      if name in birthdays:
101          del birthdays[name]
102      else:
103          print('That name is not found.')
104 
105  # Call the main function.
106  main()
```

The purpose of the `delete` function is to allow the user to delete an existing birthday from the dictionary. It accepts the dictionary as an argument. Line 97 gets a name from the user. The `if` statement in line 100 determines whether the name is in the dictionary. If so, line 101 deletes it. If the name is not in the dictionary, line 103 prints a message indicating so.

Program Output (with input shown in bold)

```
Friends and Their Birthdays
---------------------------
1. Look up a birthday
2. Add a new birthday
3. Change a birthday
```

(program output continues)

Program Output *(continued)*

```
4. Delete a birthday
5. Quit the program

Enter your choice: 2 Enter
Enter a name: Cameron Enter
Enter a birthday: 10/12/1990 Enter

Friends and Their Birthdays
---------------------------
1. Look up a birthday
2. Add a new birthday
3. Change a birthday
4. Delete a birthday
5. Quit the program

Enter your choice: 2 Enter
Enter a name: Kathryn Enter
Enter a birthday: 5/7/1989 Enter

Friends and Their Birthdays
---------------------------
1. Look up a birthday
2. Add a new birthday
3. Change a birthday
4. Delete a birthday
5. Quit the program

Enter your choice: 1 Enter
Enter a name: Cameron Enter
10/12/1990

Friends and Their Birthdays
---------------------------
1. Look up a birthday
2. Add a new birthday
3. Change a birthday
4. Delete a birthday
5. Quit the program

Enter your choice: 1 Enter
Enter a name: Kathryn Enter
5/7/1989

Friends and Their Birthdays
---------------------------
1. Look up a birthday
2. Add a new birthday
3. Change a birthday
```

```
4. Delete a birthday
5. Quit the program

Enter your choice: **3** Enter
Enter a name: **Kathryn** Enter
Enter the new birthday: **5/7/1988** Enter

Friends and Their Birthdays
---------------------------
1. Look up a birthday
2. Add a new birthday
3. Change a birthday
4. Delete a birthday
5. Quit the program

Enter your choice: **1** Enter
Enter a name: **Kathryn** Enter
5/7/1988

Friends and Their Birthdays
---------------------------
1. Look up a birthday
2. Add a new birthday
3. Change a birthday
4. Delete a birthday
5. Quit the program

Enter your choice: **4** Enter
Enter a name: **Cameron** Enter

Friends and Their Birthdays
---------------------------
1. Look up a birthday
2. Add a new birthday
3. Change a birthday
4. Delete a birthday
5. Quit the program

Enter your choice: **1** Enter
Enter a name: **Cameron** Enter
Not found.

Friends and Their Birthdays
---------------------------
1. Look up a birthday
2. Add a new birthday
3. Change a birthday
4. Delete a birthday
5. Quit the program
Enter your choice: **5** Enter
```

Checkpoint

9.1 An element in a dictionary has two parts. What are they called?

9.2 Which part of a dictionary element must be immutable?

9.3 Suppose `'start' : 1472` is an element in a dictionary. What is the key? What is the value?

9.4 Suppose a dictionary named `employee` has been created. What does the following statement do?

```
employee['id'] = 54321
```

9.5 What will the following code display?

```
stuff = {1 : 'aaa', 2 : 'bbb', 3 : 'ccc'}
print(stuff[3])
```

9.6 How can you determine whether a key-value pair exists in a dictionary?

9.7 Suppose a dictionary named `inventory` exists. What does the following statement do?

```
del inventory[654]
```

9.8 What will the following code display?

```
stuff = {1 : 'aaa', 2 : 'bbb', 3 : 'ccc'}
print(len(stuff))
```

9.9 What will the following code display?

```
stuff = {1 : 'aaa', 2 : 'bbb', 3 : 'ccc'}
for k in stuff:
    print(k)
```

9.10 What is the difference between the dictionary methods `pop` and `popitem`?

9.11 What does the `items` method return?

9.12 What does the `keys` method return?

9.13 What does the `values` method return?

9.2 Sets

CONCEPT: A set contains a collection of unique values and works like a mathematical set.

VideoNote
Introduction to Sets

A *set* is an object that stores a collection of data in the same way as mathematical sets. Here are some important things to know about sets:

- All the elements in a set must be unique. No two elements can have the same value.
- Sets are unordered, which means that the elements in a set are not stored in any particular order.
- The elements that are stored in a set can be of different data types.

Creating a Set

To create a set, you have to call the built-in `set` function. Here is an example of how you create an empty set:

```
myset = set()
```

After this statement executes, the `myset` variable will reference an empty set. You can also pass one argument to the `set` function. The argument that you pass must be an object that contains iterable elements, such as a list, a tuple, or a string. The individual elements of the object that you pass as an argument become elements of the set. Here is an example:

```
myset = set(['a', 'b', 'c'])
```

In this example we are passing a list as an argument to the `set` function. After this statement executes, the `myset` variable references a set containing the elements `'a'`, `'b'`, and `'c'`.

If you pass a string as an argument to the set function, each individual character in the string becomes a member of the set. Here is an example:

```
myset = set('abc')
```

After this statement executes, the `myset` variable will reference a set containing the elements `'a'`, `'b'`, and `'c'`.

Sets cannot contain duplicate elements. If you pass an argument containing duplicate elements to the set function, only one of the duplicated elements will appear in the set. Here is an example:

```
myset = set('aaabc')
```

The character `'a'` appears multiple times in the string, but it will appear only once in the set. After this statement executes, the `myset` variable will reference a set containing the elements `'a'`, `'b'`, and `'c'`.

What if you want to create a set in which each element is a string containing more than one character? For example, how would you create a set containing the elements `'one'`, `'two'`, and `'three'`? The following code does not accomplish the task because you can pass no more than one argument to the `set` function:

```
# This is an ERROR!
myset = set('one', 'two', 'three')
```

The following does not accomplish the task either:

```
# This does not do what we intend.
myset = set('one two three')
```

After this statement executes, the `myset` variable will reference a set containing the elements `'o'`, `'n'`, `'e'`, `' '`, `'t'`, `'w'`, `'h'`, and `'r'`. To create the set that we want, we have to pass a list containing the strings ‘`one`’, ‘`two`’, and ‘`three`’ as an argument to the set function. Here is an example:

```
# OK, this works.
myset = set(['one', 'two', 'three'])
```

After this statement executes, the `myset` variable will reference a set containing the elements `'one'`, `'two'`, and `'three'`.

Getting the Number of Elements in a Set

As with lists, tuples, and dictionaries, you can use the `len` function to get the number of elements in a set. The following interactive session demonstrates:

```
1  >>> myset = set([1, 2, 3, 4, 5]) Enter
2  >>> len(myset) Enter
3  5
4  >>>
```

Adding and Removing Elements

Sets are mutable objects, so you can add items to them and remove items from them. You use the `add` method to add an element to a set. The following interactive session demonstrates:

```
1  >>> myset = set() Enter
2  >>> myset.add(1) Enter
3  >>> myset.add(2) Enter
4  >>> myset.add(3) Enter
5  >>> myset Enter
6  {1, 2, 3}
7  >>> myset.add(2) Enter
8  >>> myset
9  {1, 2, 3}
```

The statement in line 1 creates an empty set and assigns it to the `myset` variable. The statements in lines 2 through 4 add the values 1, 2, and 3 to the set. Line 5 displays the contents of the set, which is shown in line 6.

The statement in line 7 attempts to add the value 2 to the set. The value 2 is already in the set, however. If you try to add a duplicate item to a set with the `add` method, the method does not raise an exception. It simply does not add the item.

You can add a group of elements to a set all at one time with the `update` method. When you call the `update` method as an argument, you pass an object that contains iterable elements, such as a list, a tuple, string, or another set. The individual elements of the object that you pass as an argument become elements of the set. The following interactive session demonstrates:

```
1  >>> myset = set([1, 2, 3]) Enter
2  >>> myset.update([4, 5, 6]) Enter
3  >>> myset Enter
4  {1, 2, 3, 4, 5, 6}
5  >>>
```

The statement in line 1 creates a set containing the values 1, 2, and 3. Line 2 adds the values 4, 5, and 6. The following session shows another example:

```
1  >>> set1 = set([1, 2, 3]) Enter
2  >>> set2 = set([8, 9, 10]) Enter
3  >>> set1.update(set2) Enter
4  >>> set1
```

```
5  {1, 2, 3, 8, 9, 10}
6  >>> set2 Enter
7  {8, 9, 10}
8  >>>
```

Line 1 creates a set containing the values 1, 2, and 3 and assigns it to the `set1` variable. Line 2 creates a set containing the values 8, 9, and 10 and assigns it to the `set2` variable. Line 3 calls the `set1.update` method, passing `set2` as an argument. This causes the element of `set2` to be added to `set1`. Notice that `set2` remains unchanged. The following session shows another example:

```
1  >>> myset = set([1, 2, 3]) Enter
2  >>> myset.update('abc') Enter
3  >>> myset Enter
4  {'a', 1, 2, 3, 'c', 'b'}
5  >>>
```

The statement in line 1 creates a set containing the values 1, 2, and 3. Line 2 calls the `myset.update` method, passing the string `'abc'` as an argument. This causes the each character of the string to be added as an element to `myset`.

You can remove an item from a set with either the `remove` method or the `discard` method. You pass the item that you want to remove as an argument to either method, and that item is removed from the set. The only difference between the two methods is how they behave when the specified item is not found in the set. The `remove` method raises a `KeyError` exception, but the `discard` method does not raise an exception. The following interactive session demonstrates:

```
 1  >>> myset = set([1, 2, 3, 4, 5]) Enter
 2  >>> myset Enter
 3  {1, 2, 3, 4, 5}
 4  >>> myset.remove(1) Enter
 5  >>> myset Enter
 6  {2, 3, 4, 5}
 7  >>> myset.discard(5) Enter
 8  >>> myset Enter
 9  {2, 3, 4}
10  >>> myset.discard(99) Enter
11  >>> myset.remove(99) Enter
12  Traceback (most recent call last):
13    File "<pyshell#12>", line 1, in <module>
14      myset.remove(99)
15  KeyError: 99
16  >>>
```

Line 1 creates a set with the elements 1, 2, 3, 4, and 5. Line 2 displays the contents of the set, which is shown in line 3. Line 4 calls the `remove` method to remove the value 1 from the set. You can see in the output shown in line 6 that the value 1 is no longer in the set. Line 7 calls the `discard` method to remove the value 5 from the set. You can see in the output in line 9 that the value 5 is no longer in the set. Line 10 calls the `discard` method to remove the value 99 from the set. The value is not found in the set, but the `discard` method does not raise an exception. Line 11 calls the `remove` method to remove the value 99 from the set. Because the value is not in the set, a `KeyError` exception is raised, as shown in lines 12 through 15.

You can clear all the elements of a set by calling the `clear` method. The following interactive session demonstrates:

```
1  >>> myset = set([1, 2, 3, 4, 5]) Enter
2  >>> myset Enter
3  {1, 2, 3, 4, 5}
4  >>> myset.clear() Enter
5  >>> myset Enter
6  set()
7  >>>
```

The statement in line 4 calls the `clear` method to clear the set. Notice in line 6 that when we display the contents of an empty set, the interpreter displays `set()`.

Using the `for` Loop to Iterate over a Set

You can use the `for` loop in the following general format to iterate over all the elements in a set:

```
for var in set:
    statement
    statement
    etc.
```

In the general format, *`var`* is the name of a variable and *`set`* is the name of a set. This loop iterates once for each element in the set. Each time the loop iterates, *`var`* is assigned an element. The following interactive session demonstrates:

```
1  >>> myset = set(['a', 'b', 'c']) Enter
2  >>> for val in myset: Enter
3          print(val) Enter Enter
4
5  a
6  c
7  b
8  >>>
```

Lines 2 through 3 contain a `for` loop that iterates once for each element of the `myset` set. Each time the loop iterates, an element of the set is assigned to the `val` variable. Line 3 prints the value of the `val` variable. Lines 5 through 7 show the output of the loop.

Using the `in` and `not in` Operators to Test for a Value in a Set

You can use the `in` operator to determine whether a value exists in a set. The following interactive session demonstrates:

```
1  >>> myset = set([1, 2, 3]) Enter
2  >>> if 1 in myset: Enter
3          print('The value 1 is in the set.') Enter Enter
4
5  The value 1 is in the set.
6  >>>
```

The `if` statement in line 2 determines whether the value 1 is in the `myset` set. If it is, the statement in line 3 displays a message.

You can also use the `not in` operator to determine if a value does not exist in a set, as demonstrated in the following session:

```
1  >>> myset = set([1, 2, 3]) Enter
2  >>> if 99 not in myset: Enter
3          print('The value 99 is not in the set.') Enter Enter
4
5  The value 99 is not in the set.
6  >>>
```

Finding the Union of Sets

The union of two sets is a set that contains all the elements of both sets. In Python, you can call the `union` method to get the union of two sets. Here is the general format:

```
set1.union(set2)
```

In the general format, *`set1`* and *`set2`* are sets. The method returns a set that contains the elements of both *`set1`* and *`set2`*. The following interactive session demonstrates:

```
1  >>> set1 = set([1, 2, 3, 4]) Enter
2  >>> set2 = set([3, 4, 5, 6]) Enter
3  >>> set3 = set1.union(set2) Enter
4  >>> set3 Enter
5  {1, 2, 3, 4, 5, 6}
6  >>>
```

The statement in line 3 calls the `set1` object's `union` method, passing `set2` as an argument. The method returns a set that contains all the elements of `set1` and `set2` (without duplicates, of course). The resulting set is assigned to the `set3` variable.

You can also use the | operator to find the union of two sets. Here is the general format of an expression using the | operator with two sets:

```
set1 | set2
```

In the general format, *`set1`* and *`set2`* are sets. The expression returns a set that contains the elements of both *`set1`* and *`set2`*. The following interactive session demonstrates:

```
1  >>> set1 = set([1, 2, 3, 4]) Enter
2  >>> set2 = set([3, 4, 5, 6]) Enter
3  >>> set3 = set1 | set2 Enter
4  >>> set3 Enter
5  {1, 2, 3, 4, 5, 6}
6  >>>
```

Finding the Intersection of Sets

The intersection of two sets is a set that contains only the elements that are found in both sets. In Python, you can call the `intersection` method to get the intersection of two sets. Here is the general format:

```
set1.intersection(set2)
```

In the general format, *set1* and *set2* are sets. The method returns a set that contains the elements that are found in both *set1* and *set2*. The following interactive session demonstrates:

```
1  >>> set1 = set([1, 2, 3, 4]) Enter
2  >>> set2 = set([3, 4, 5, 6]) Enter
3  >>> set3 = set1.intersection(set2) Enter
4  >>> set3 Enter
5  {3, 4}
6  >>>
```

The statement in line 3 calls the `set1` object's `intersection` method, passing `set2` as an argument. The method returns a set that contains the elements that are found in both `set1` and `set2`. The resulting set is assigned to the `set3` variable.

You can also use the `&` operator to find the intersection of two sets. Here is the general format of an expression using the `&` operator with two sets:

```
set1 & set2
```

In the general format, *set1* and *set2* are sets. The expression returns a set that contains the elements that are found in both *set1* and *set2*. The following interactive session demonstrates:

```
1  >>> set1 = set([1, 2, 3, 4]) Enter
2  >>> set2 = set([3, 4, 5, 6]) Enter
3  >>> set3 = set1 & set2 Enter
4  >>> set3 Enter
5  {3, 4}
6  >>>
```

Finding the Difference of Sets

The difference of `set1` and `set2` is the elements that appear in `set1` but do not appear in `set2`. In Python, you can call the `difference` method to get the difference of two sets. Here is the general format:

```
set1.difference(set2)
```

In the general format, *set1* and *set2* are sets. The method returns a set that contains the elements that are found in *set1* but not in *set2*. The following interactive session demonstrates:

```
1  >>> set1 = set([1, 2, 3, 4]) Enter
2  >>> set2 = set([3, 4, 5, 6]) Enter
3  >>> set3 = set1.difference(set2) Enter
4  >>> set3 Enter
5  {1, 2}
6  >>>
```

You can also use the − operator to find the difference of two sets. Here is the general format of an expression using the − operator with two sets:

```
set1 - set2
```

In the general format, *set1* and *set2* are sets. The expression returns a set that contains the elements that are found in *set1* but not in *set2*. The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4]) Enter
>>> set2 = set([3, 4, 5, 6]) Enter
>>> set3 = set1 - set2 Enter
>>> set3 Enter
{1, 2}
>>>
```

Finding the Symmetric Difference of Sets

The symmetric difference of two sets is a set that contains the elements that are not shared by the sets. In other words, it is the elements that are in one set but not in both. In Python, you can call the `symmetric_difference` method to get the symmetric difference of two sets. Here is the general format:

```
set1.symmetric_difference(set2)
```

In the general format, *set1* and *set2* are sets. The method returns a set that contains the elements that are found in either *set1* or *set2* but not both sets. The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4]) Enter
>>> set2 = set([3, 4, 5, 6]) Enter
>>> set3 = set1.symmetric_difference(set2) Enter
>>> set3 Enter
{1, 2, 5, 6}
>>>
```

You can also use the ^ operator to find the symmetric difference of two sets. Here is the general format of an expression using the ^ operator with two sets:

```
set1 ^ set2
```

In the general format, *set1* and *set2* are sets. The expression returns a set that contains the elements that are found in either *set1* or *set2* but not both sets. The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4]) Enter
>>> set2 = set([3, 4, 5, 6]) Enter
>>> set3 = set1 ^ set2 Enter
>>> set3 Enter
{1, 2, 5, 6}
>>>
```

Finding Subsets and Supersets

Suppose you have two sets and one of those sets contains all of the elements of the other set. Here is an example:

```
set1 = set([1, 2, 3, 4])
set2 = set([2, 3])
```

In this example, `set1` contains all the elements of `set2`, which means that `set2` is a *subset* of `set1`. It also means that `set1` is a *superset* of `set2`. In Python, you can call the `issubset` method to determine whether one set is a subset of another. Here is the general format:

```
set2.issubset(set1)
```

In the general format, *`set1`* and *`set2`* are sets. The method returns `True` if `set2` is a subset of `set1`. Otherwise, it returns `False`. You can call the `issuperset` method to determine whether one set is a superset of another. Here is the general format:

```
set1.issuperset(set2)
```

In the general format, *`set1`* and *`set2`* are sets. The method returns `True` if `set1` is a superset of `set2`. Otherwise, it returns `False`. The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4]) Enter
>>> set2 = set([2, 3]) Enter
>>> set2.issubset(set1) Enter
True
>>> set1.issuperset(set2) Enter
True
>>>
```

You can also use the <= operator to determine whether one set is a subset of another and the >= operator to determine whether one set is a superset of another. Here is the general format of an expression using the <= operator with two sets:

```
set2 <= set1
```

In the general format, *`set1`* and *`set2`* are sets. The expression returns `True` if `set2` is a subset of `set1`. Otherwise, it returns `False`. Here is the general format of an expression using the >= operator with two sets:

```
set1 >= set2
```

In the general format, *`set1`* and *`set2`* are sets. The expression returns `True` if `set1` is a subset of `set2`. Otherwise, it returns `False`. The following interactive session demonstrates:

```
>>> set1 = set([1, 2, 3, 4]) Enter
>>> set2 = set([2, 3]) Enter
>>> set2 <= set1 Enter
True
>>> set1 >= set2 Enter
True
>>> set1 <= set2 Enter
False
```

In the Spotlight:

Set Operations

In this section you will look at Program 9-3, which demonstrates various set operations. The program creates two sets: one that holds the names of students on the baseball team and another that holds the names of students on the basketball team. The program then performs the following operations:

- It finds the intersection of the sets to display the names of students who play both sports.
- It finds the union of the sets to display the names of students who play either sport.
- It finds the difference of the baseball and basketball sets to display the names of students who play baseball but not basketball.
- It finds the difference of the basketball and baseball (*basketball – baseball*) sets to display the names of students who play basketball but not baseball. It also finds the difference of the baseball and basketball (*baseball – basketball*) sets to display the names of students who play baseball but not basketball.
- It finds the symmetric difference of the basketball and baseball sets to display the names of students who play one sport but not both.

Program 9-3 (sets.py)

```
# This program demonstrates various set operations.
baseball = set(['Jodi', 'Carmen', 'Aida', 'Alicia'])
basketball = set(['Eva', 'Carmen', 'Alicia', 'Sarah'])

# Display members of the baseball set.
print('The following students are on the baseball team:')
for name in baseball:
    print(name)

# Display members of the basketball set.
print()
print('The following students are on the basketball team:')
for name in basketball:
    print(name)

# Demonstrate intersection
print()
print('The following students play both baseball and basketball:')
for name in baseball.intersection(basketball):
    print(name)

# Demonstrate union
print()
print('The following students play either baseball or basketball:')
```

(program continues)

Program 9-3 *(continued)*

```
for name in baseball.union(basketball):
    print(name)

# Demonstrate difference of baseball and basketball
print()
print('The following students play baseball, but not basketball:')
for name in baseball.difference(basketball):
    print(name)

# Demonstrate difference of basketball and baseball
print()
print('The following students play basketball, but not baseball:')
for name in basketball.difference(baseball):
    print(name)

# Demonstrate symmetric difference
print()
print('The following students play one sport, but not both:')
for name in baseball.symmetric_difference(basketball):
    print(name)
```

Program Output

```
The following students are on the baseball team:
Jodi
Aida
Carmen
Alicia

The following students are on the basketball team:
Sarah
Eva
Alicia
Carmen

The following students play both baseball and basketball:
Alicia
Carmen

The following students play either baseball or basketball:
Sarah
Alicia
Jodi
Eva
Aida
Carmen

The following students play baseball but not basketball:
Jodi
Aida
```

```
The following students play basketball but not baseball:
Sarah
Eva

The following students play one sport but not both:
Sarah
Aida
Jodi
Eva
```

Checkpoint

9.14 Are the elements of a set ordered or unordered?

9.15 Does a set allow you to store duplicate elements?

9.16 How do you create an empty set?

9.17 After the following statement executes, what elements will be stored in the `myset` set?

```
myset = set('Jupiter')
```

9.18 After the following statement executes, what elements will be stored in the `myset` set?

```
myset = set(25)
```

9.19 After the following statement executes, what elements will be stored in the `myset` set?

```
myset = set('www xxx yyy zzz')
```

9.20 After the following statement executes, what elements will be stored in the `myset` set?

```
myset = set([1, 2, 2, 3, 4, 4, 4])
```

9.21 After the following statement executes, what elements will be stored in the `myset` set?

```
myset = set(['www', 'xxx', 'yyy', 'zzz'])
```

9.22 How do you determine the number of elements in a set?

9.23 After the following statement executes, what elements will be stored in the `myset` set?

```
myset = set([10, 9, 8])
myset.update([1, 2, 3])
```

9.24 After the following statement executes, what elements will be stored in the `myset` set?

```
myset = set([10, 9, 8])
myset.update('abc')
```

9.25 What is the difference between the `remove` and `discard` methods?

9.26 How can you determine whether a specific element exists in a set?

9.27 After the following code executes, what elements will be members of `set3`?

```
set1 = set([10, 20, 30])
set2 = set([100, 200, 300])
set3 = set1.union(set2)
```

9.28 After the following code executes, what elements will be members of `set3`?

```
set1 = set([1, 2, 3, 4])
set2 = set([3, 4, 5, 6])
set3 = set1.intersection(set2)
```

9.29 After the following code executes, what elements will be members of `set3`?

```
set1 = set([1, 2, 3, 4])
set2 = set([3, 4, 5, 6])
set3 = set1.difference(set2)
```

9.30 After the following code executes, what elements will be members of `set3`?

```
set1 = set([1, 2, 3, 4])
set2 = set([3, 4, 5, 6])
set3 = set2.difference(set1)
```

9.31 After the following code executes, what elements will be members of `set3`?

```
set1 = set(['a', 'b', 'c'])
set2 = set(['b', 'c', 'd'])
set3 = set1.symmetric_difference(set2)
```

9.32 Look at the following code:

```
set1 = set([1, 2, 3, 4])
set2 = set([2, 3])
```

Which of the sets is a subset of the other?

Which of the sets is a superset of the other?

9.3 Serializing Objects

CONCEPT: Serializing a object is the process of converting the object to a stream of bytes that can be saved to a file for later retrieval. In Python, object serialization is called pickling.

In Chapter 6 you learned how to store data in a text file. Sometimes you need to store the contents of a complex object, such as a dictionary or a set, to a file. The easiest way to save an object to a file is to serialize the object. When an object is *serialized*, it is converted to a stream of bytes that can be easily stored in a file for later retrieval.

In Python, the process of serializing an object is referred to as *pickling*. The Python standard library provides a module named `pickle` that has various functions for serializing, or pickling, objects.

Once you import the `pickle` module, you perform the following steps to pickle an object:

- You open a file for binary writing.
- You call the `pickle` module's `dump` method to pickle the object and write it to the specified file.
- After you have pickled all the objects that you want to save to the file, you close the file.

Let's take a more detailed look at these steps. To open a file for binary writing, you use `'wb'` as the mode when you call the `open` function. For example, the following statement opens a file named `mydata.dat` for binary writing:

```
outputfile = open('mydata.dat', 'wb')
```

Once you have opened a file for binary writing, you call the `pickle` module's `dump` function. Here is the general format of the `dump` method:

```
pickle.dump(object, file)
```

In the general format, *`object`* is a variable that references the object you want to pickle, and *`file`* is a variable that references a file object. After the function executes, the object referenced by *`object`* will be serialized and written to the file. (You can pickle just about any type of object, including lists, tuples, dictionaries, sets, strings, integers, and floating-point numbers.)

You can save as many pickled objects as you want to a file. When you are finished, you call the file object's `close` method to close the file. The following interactive session provides a simple demonstration of pickling a dictionary:

```
1  >>> import pickle Enter
2  >>> phonebook = {'Chris' : '555-1111', Enter
3                  'Katie' : '555-2222', Enter
4                  'Joanne' : '555-3333'} Enter
5  >>> output_file = open('phonebook.dat', 'wb') Enter
6  >>> pickle.dump(phonebook, output_file) Enter
7  >>> output_file.close() Enter
8  >>>
```

Let's take a closer look at the session:

- Line 1 imports the `pickle` module.
- Lines 2 through 4 create a dictionary containing names (as keys) and phone numbers (as values).
- Line 5 opens a file named `phonebook.dat` for binary writing.
- Line 6 calls the `pickle` module's `dump` function to serialize the `phonebook` dictionary and write it to the `phonebook.dat` file.
- Line 7 closes the `phonebook.dat` file.

At some point, you will need to retrieve, or unpickle, the objects that you have pickled. Here are the steps that you perform:

- You open a file for binary reading.
- You call the `pickle` module's `load` function to retrieve an object from the file and unpickle it.
- After you have unpickled all the objects that you want from the file, you close the file.

Let's take a more detailed look at these steps. To open a file for binary reading, you use `'rb'` as the mode when you call the `open` function. For example, the following statement opens a file named `mydata.dat` for binary reading:

```
inputfile = open('mydata.dat', 'rb')
```

Once you have opened a file for binary reading, you call the `pickle` module's `load` function. Here is the general format of a statement that calls the `load` function:

```
object = pickle.load(file)
```

In the general format, *`object`* is a variable, and *`file`* is a variable that references a file object. After the function executes, the *`object`* variable will reference an object that was retrieved from the file and unpickled.

You can unpickle as many objects as necessary from the file. (If you try to read past the end of the file, the load function will raise an `EOFError` exception.) When you are finished, you call the file object's `close` method to close the file. The following interactive session provides a simple demonstration of unpickling the `phonebook` dictionary that was pickled in the previous session:

```
1  >>> import pickle Enter
2  >>> input_file = open('phonebook.dat', 'rb') Enter
3  >>> pb = pickle.load(inputfile) Enter
4  >>> pb Enter
5  {'Chris': '555-1111', 'Joanne': '555-3333', 'Katie': '555-2222'}
6  >>> input_file.close() Enter
7  >>>
```

Let's take a closer look at the session:

- Line 1 imports the `pickle` module.
- Line 2 opens a file named `phonebook.dat` for binary reading.
- Line 3 calls the `pickle` module's `load` function to retrieve and unpickle an object from the `phonebook.dat` file. The resulting object is assigned to the `pb` variable.
- Line 4 displays the dictionary referenced by the `pb` variable. The output is shown in line 5.
- Line 6 closes the `phonebook.dat` file.

Program 9-4 shows an example program that demonstrates object pickling. It prompts the user to enter personal information (name, age, and weight) about as many people as he or she wishes. Each time the user enters information about a person, the information is stored in a dictionary, and then the dictionary is pickled and saved to a file named `info.dat`. After the program has finished, the `info.dat` file will hold one pickled dictionary object for every person about whom the user entered information.

Program 9-4 (pickle_objects.py)

```
# This program demonstrates object pickling.
import pickle

# main function
```

```
def main():
    again = 'y' # To control loop repetition

    # Open a file for binary writing.
    output_file = open('info.dat', 'wb')

    # Get data until the user wants to stop.
    while again.lower() == 'y':
        # Get data about a person and save it.
        save_data(output_file)

        # Does the user want to enter more data?
        again = input('Enter more data? (y/n): ')

    # Close the file.
    output_file.close()

# The save_data function gets data about a person,
# stores it in a dictionary, and then pickles the
# dictionary to the specified file.
def save_data(file):
    # Create an empty dictionary.
    person = {}

    # Get data for a person and store
    # it in the dictionary.
    person['name'] = input('Name: ')
    person['age'] = int(input('Age: '))
    person['weight'] = float(input('Weight: '))

    # Pickle the dictionary.
    pickle.dump(person, file)

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Name: Angie Enter
Age: 25 Enter
Weight: 122 Enter
Enter more data? (y/n): y Enter
Name: Carl Enter
Age: 28 Enter
Weight: 175 Enter
Enter more data? (y/n): n Enter
```

Let's take a closer look at the main function:

- The again variable that is initialized in line 6 is used to control loop repetitions.
- Line 9 opens the file info.dat for binary writing. The file object is assigned to the output_file variable.
- The while loop that begins in line 12 repeats as long as the again variable references 'y' or 'Y'.
- Inside the while loop, line 14 calls the save_data function, passing the output_file variable as an argument. The purpose of the save_data function is to get data about a person and save it to the file as a pickled dictionary object.
- Line 17 prompts the user to enter y or n to indicate whether he or she wants to enter more data. The input is assigned to the again variable.
- Outside the loop, line 20 closes the file.

Now, let's look at the save_data function:

- Line 27 creates an empty dictionary, referenced by the person variable.
- Line 31 prompts the user to enter the person's name and stores the input in the person dictionary. After this statement executes, the dictionary will contain a key-value pair that has the string 'name' as the key and the user's input as the value.
- Line 32 prompts the user to enter the person's age and stores the input in the person dictionary. After this statement executes, the dictionary will contain a key-value pair that has the string 'age' as the key and the user's input, as an int, as the value.
- Line 33 prompts the user to enter the person's weight and stores the input in the person dictionary. After this statement executes, the dictionary will contain a key-value pair that has the string 'weight' as the key and the user's input, as a float, as the value.
- Line 36 pickles the person dictionary and writes it to the file.

Program 9-5 demonstrates how the dictionary objects that have been pickled and saved to the info.dat file can be retrieved and unpickled.

Program 9-5 (unpickle_objects.py)

```
# This program demonstrates object unpickling.
import pickle

# main function
def main():
    end_of_file = False # To indicate end of file

    # Open a file for binary reading.
    input_file = open('info.dat', 'rb')

    # Read to the end of the file.
    while not end_of_file:
        try:
            # Unpickle the next object.
            person = pickle.load(input_file)

```

```
17                # Display the object.
18                display_data(person)
19            except EOFError:
20                # Set the flag to indicate the end
21                # of the file has been reached.
22                end_of_file = True
23
24        # Close the file.
25        input_file.close()
26
27    # The display_data function displays the person data
28    # in the dictionary that is passed as an argument.
29    def display_data(person):
30        print('Name:', person['name'])
31        print('Age:', person['age'])
32        print('Weight:', person['weight'])
33        print()
34
35    # Call the main function.
36    main()
```

Program Output

```
Name: Angie
Age: 25
Weight: 122.0

Name: Carl
Age: 28
Weight: 175.0
```

Let's take a closer look at the `main` function:

- The `end_of_file` variable that is initialized in line 6 is used to indicate when the program has reached the end of the `info.dat` file. Notice that the variable is initialized with the Boolean value `False`.
- Line 9 opens the file `info.dat` for binary reading. The file object is assigned to the `input_file` variable.
- The `while` loop that begins in line 12 repeats as long as `end_of_file` is `False`.
- Inside the `while` loop, a `try/except` statement appears in lines 13 through 22.
- Inside the try suite, line 15 reads an object from the file, unpickles it, and assigns it to the `person` variable. If the end of the file has already been reached, this statement raises an `EOFError` exception, and the program jumps to the `except` clause in 19. Otherwise, line 18 calls the `display_data` function, passing the `person` variable as an argument.
- When an `EOFError` exception occurs, line 22 sets the `end_of_file` variable to `True`. This causes the `while` loop to stop iterating.