CHAPTER 8

More About Strings

TOPICS

8.1 Basic String Operations

CONCEPT: **Python provides several ways to access the individual characters in a string. Strings also have methods that allow you to perform operations on them.**

Many of the programs that you have written so far have worked with strings, but only in a limited way. The operations that you have performed with strings so far have primarily involved only input and output. For example, you have read strings as input from the keyboard and from files and sent strings as output to the screen and to files.

There are many types of programs that not only read strings as input and write strings as output, but also perform operations on strings. Word processing programs, for example, manipulate large amounts of text and thus work extensively with strings. Email programs and search engines are other examples of programs that perform operations on strings.

Python provides a wide variety of tools and programming techniques that you can use to examine and manipulate strings. In fact, strings are a type of sequence, so many of the concepts that you learned about sequences in Chapter 7 apply to strings as well. We will look at many of these in this chapter.

Accessing the Individual Characters in a String

Some programming tasks require that you access the individual characters in a string. For example, you are probably familiar with websites that require you to set up a password. For security reasons, many sites require that your password have at least one uppercase letter, at least one lowercase letter, and at least one digit. When you set up your password, a program examines each character to ensure that the password meets these qualifications. (Later in this chapter you will see an example of a program that does this sort of thing.) In this section we will look at two techniques that you can use in Python to access the individual characters in a string: using the `for` loop, and indexing.

Iterating over a String with the `for` Loop

One of the easiest ways to access the individual characters in a string is to use the `for` loop. Here is the general format:

```
for variable in string:
    statement
    statement
    etc.
```

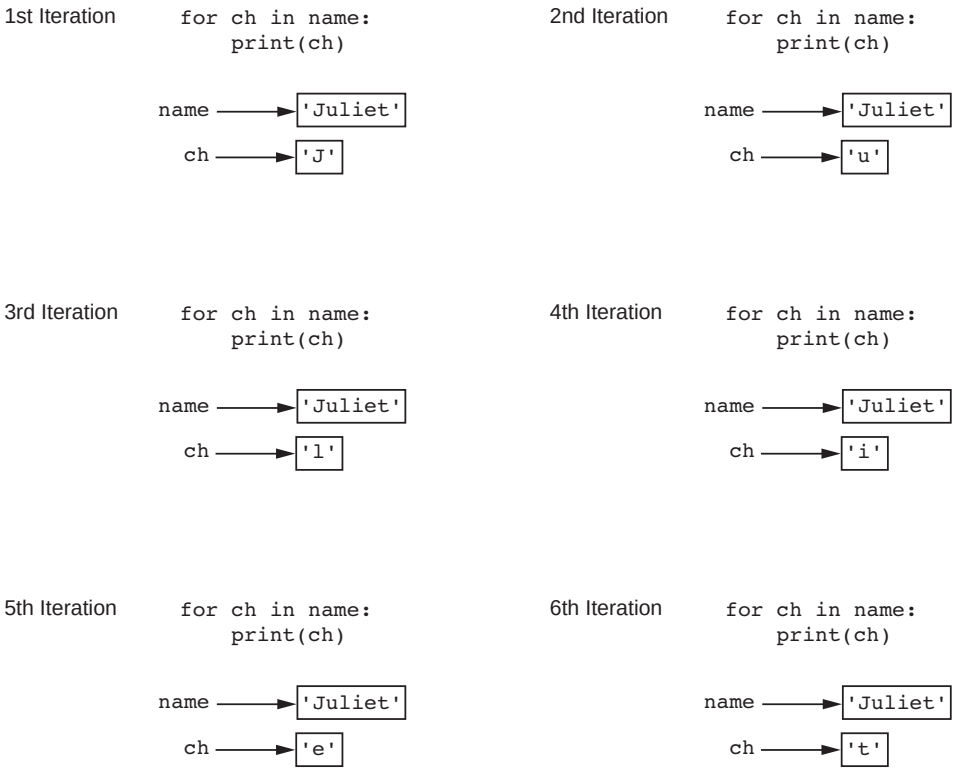
In the general format, *`variable`* is the name of a variable and *`string`* is either a string literal or a variable that references a string. Each time the loop iterates, *`variable`* will reference a copy of a character in *`string`*, beginning with the first character. We say that the loop iterates over the characters in the string. Here is an example:

```
name = 'Juliet'
for ch in name:
    print(ch)
```

The `name` variable references a string with six characters, so this loop will iterate six times. The first time the loop iterates, the `ch` variable will reference `'J'`, the second time the loop iterates the `ch` variable will reference `'u'`, and so forth. This is illustrated in Figure 8-1. When the code executes, it will display the following:

```
J
u
l
i
e
t
```

Figure 8-1 Iterating over the string `'Juliet'`



NOTE: Figure 8-1 illustrates how the `ch` variable references a copy of a character from the string as the loop iterates. If we change the value that `ch` references in the loop, it has no effect on the string referenced by `name`. To demonstrate, look at the following:

```
1 name = 'Juliet'
2 for ch in name:
3       ch = 'X'
4       print(name)
```

The statement in line 3 merely reassigns the `ch` variable to a different value each time the loop iterates. It has no effect on the string `'Juliet'` that is referenced by `name`, and it has no effect on the number of times the loop iterates. When this code executes, the statement in line 4 will print:

```
Juliet
```

Program 8-1 shows another example. This program asks the user to enter a string. It then uses a `for` loop to iterate over the string, counting the number of times that the letter T (uppercase or lowercase) appears.

Program 8-1 (count_Ts.py)

```
1  # This program counts the number of times
2  # the letter T (uppercase or lowercase)
3  # appears in a string.
4
5  def main():
6      # Create a variable to use to hold the count.
7      # The variable must start with 0.
8      count = 0
9
10     # Get a string from the user.
11     my_string = input('Enter a sentence: ')
12
13     # Count the Ts.
14     for ch in my_string:
15         if ch == 'T' or ch == 't':
16             count += 1
17
18     # Print the result.
19     print('The letter T appears', count, 'times.')
20
21 # Call the main function.
22 main()
```

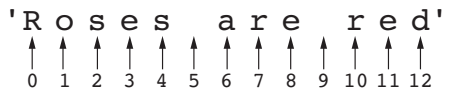
Program Output (with input shown in bold)

```
Enter a sentence: Today we sold twenty-two toys. Enter
The letter T appears 5 times.
```

Indexing

Another way that you can access the individual characters in a string is with an index. Each character in a string has an index that specifies its position in the string. Indexing starts at 0, so the index of the first character is 0, the index of the second character is 1, and so forth. The index of the last character in a string is 1 less than the number of characters in the string. Figure 8-2 shows the indexes for each character in the string `'Roses are red'`. The string has 13 characters, so the character indexes range from 0 through 12.

Figure 8-2 String indexes

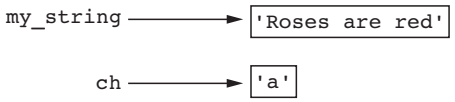


You can use an index to retrieve a copy of an individual character in a string, as shown here:

```
my_string = 'Roses are red'
ch = my_string[6]
```

The expression `my_string[6]` in the second statement returns a copy of the character at index 6 in `my_string`. After this statement executes, `ch` will reference `'a'` as shown in Figure 8-3.

Figure 8-3 Getting a copy of a character from a string



Here is another example:

```
my_string = 'Roses are red'
print(my_string[0], my_string[6], my_string[10])
```

This code will print the following:

```
R a r
```

You can also use negative numbers as indexes, to identify character positions relative to the end of the string. The Python interpreter adds negative indexes to the length of the string to determine the character position. The index –1 identifies the last character in a string, –2 identifies the next to last character, and so forth. The following code shows an example:

```
my_string = 'Roses are red'
print(my_string[-1], my_string[-2], my_string[-13])
```

This code will print the following:

```
d e R
```

`IndexError` Exceptions

An `IndexError` exception will occur if you try to use an index that is out of range for a particular string. For example, the string `'Boston'` has 6 characters, so the valid indexes are 0 through 5. (The valid negative indexes are –1 through –6.) The following is an example of code that causes an `IndexError` exception.

```
city = 'Boston'
print(city[6])
```

This type of error is most likely to happen when a loop incorrectly iterates beyond the end of a string, as shown here:

```
city = 'Boston'
index = 0
while index < 7:
    print(city[index])
    index += 1
```

The last time that this loop iterates, the `index` variable will be assigned the value 6, which is an invalid index for the string `'Boston'`. As a result, the `print` function will cause an `IndexError` exception to be raised.

The `len` Function

In Chapter 7 you learned about the `len` function, which returns the length of a sequence. The `len` function can also be used to get the length of a string. The following code demonstrates:

```
city = 'Boston'
size = len(city)
```

The second statement calls the `len` function, passing the `city` variable as an argument. The function returns the value 6, which is the length of the string `'Boston'`. This value is assigned to the `size` variable.

The `len` function is especially useful to prevent loops from iterating beyond the end of a string, as shown here:

```
city = 'Boston'
index = 0
while index < len(city):
    print(city[index])
    index += 1
```

Notice that the loop iterates as long as `index` is *less than* the length of the string. This is because the index of the last character in a string is always 1 less than the length of the string.

String Concatenation

A common operation that performed on strings is *concatenation*, or appending one string to the end of another string. You have seen examples in earlier chapters that use the + operator to concatenate strings. The + operator produces a string that is the combination of the two strings used as its operands. The following interactive session demonstrates:

```
1  >>> message = 'Hello ' + 'world' Enter
2  >>> print(message) Enter
3  Hello world
4  >>>
```

Line 1 concatenates the strings `'Hello'` and `'world'` to produce the string `'Hello world'`. The string `'Hello world'` is then assigned to the `message` variable. Line 2 prints the string that is referenced by the `message` variable. The output us shown in line 3.

Here is another interactive session that demonstrates concatenation:

```
1  >>> first_name = 'Emily' Enter
2  >>> last_name = 'Yeager' Enter
3  >>> full_name = first_name + ' ' + last_name Enter
4  >>> print(full_name) Enter
5  Emily Yeager
6  >>>
```

Line 1 assigns the string `'Emily'` to the `first_name` variable. Line 2 assigns the string `'Yeager'` to the `last_name` variable. Line 3 produces a string that is the concatenation of `first_name`, followed by a space, followed by `last_name`. The resulting string is assigned to the `full_name` variable. Line 4 prints the string referenced by `full_name`. The output is shown in line 5.

You can also use the += operator to perform concatenation. The following interactive session demonstrates:

```
1  >>> letters = 'abc' Enter
2  >>> letters += 'def' Enter
3  >>> print(letters) Enter
4  abcdef
5  >>>
```

The statement in line 2 performs string concatenation. It works the same as:

```
letters = letters + 'def'
```

After the statement in line 2 executes, the `letters` variable will reference the string `'abcdef'`. Here is another example:

```
>>> name = 'Kelly' Enter          # name is 'Kelly'
>>> name += ' ' Enter             # name is 'Kelly '
>>> name += 'Yvonne' Enter        # name is 'Kelly Yvonne'
>>> name += ' ' Enter             # name is 'Kelly Yvonne '
>>> name += 'Smith' Enter         # name is 'Kelly Yvonne Smith'
>>> print(name) Enter
Kelly Yvonne Smith
>>>
```

Keep in mind that the operand on the left side of the += operator must be an existing variable. If you specify a nonexistent variable, an exception is raised.

Strings Are Immutable

In Python, strings are immutable, which means that once they are created, they cannot be changed. Some operations, such as concatenation, give the impression that they modify strings, but in reality they do not. For example, look at Program 8-2.

Program 8-2 (concatenate.py)

```
1  # This program concatenates strings.
2
3  def main():
4      name = 'Carmen'
5      print('The name is', name)
6      name = name + ' Brown'
7      print('Now the name is', name)
8
9  # Call the main function.
10 main()
```

Program Output

```
The name is Carmen
Now the name is Carmen Brown
```

The statement in line 4 assigns the string `'Carmen'` to the `name` variable, as shown in Figure 8-4. The statement in line 6 concatenates the string `' Brown'` to the string `'Carmen'` and assigns the result to the `name` variable, as shown in Figure 8-5. As you can see from the figure, the original string `'Carmen'` is not modified. Instead, a new string containing `'Carmen Brown'` is created and assigned to the `name` variable. (The original string, `'Carmen'` is no longer usable because no variable references it. The Python interpreter will eventually remove the unusable string from memory.)

Figure 8-4 The string 'Carmen' assigned to `name`

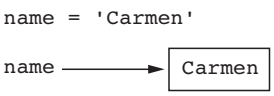
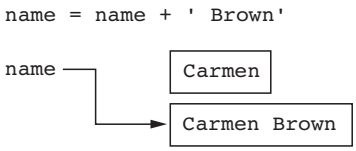


Figure 8-5 The string 'Carmen Brown' assigned to `name`



Because strings are immutable, you cannot use an expression in the form *`string[index]`* on the left side of an assignment operator. For example, the following code will cause an error:

```
# Assign 'Bill' to friend.
friend = 'Bill'
# Can we change the first character to 'J'?
friend[0] = 'J'     # No, this will cause an error!
```

The last statement in this code will raise an exception because it attempts to change the value of the first character in the string `'Bill'`.

Checkpoint

8.1 Assume the variable `name` references a string. Write a `for` loop that prints each character in the string.

8.2 What is the index of the first character in a string?

8.3 If a string has 10 characters, what is the index of the last character?

8.4 What happens if you try to use an invalid index to access a character in a string?

8.5 How do you find the length of a string?

8.6 What is wrong with the following code?

```
animal = 'Tiger'
animal[0] = 'L'
```

8.2 String Slicing

CONCEPT: **You can use slicing expressions to select a range of characters from a string**

You learned in Chapter 7 that a slice is a span of items that are taken from a sequence. When you take a slice from a string, you get a span of characters from within the string. String slices are also called *substrings*.

To get a slice of a string, you write an expression in the following general format:

```
string[start : end]
```

In the general format, *start* is the index of the first character in the slice, and *end* is the index marking the end of the slice. The expression will return a string containing a copy of the characters from *start* up to (but not including) *end*. For example, suppose we have the following:

```
full_name = 'Patty Lynn Smith'
middle_name = full_name[6:10]
```

The second statement assigns the string `'Lynn'` to the `middle_name` variable. If you leave out the *start* index in a slicing expression, Python uses 0 as the starting index. Here is an example:

```
full_name = 'Patty Lynn Smith'
first_name = full_name[:5]
```

The second statement assigns the string `'Patty'` to `first_name`. If you leave out the *end* index in a slicing expression, Python uses the length of the string as the *end* index. Here is an example:

```
full_name = 'Patty Lynn Smith'
last_name = full_name[11:]
```

The second statement assigns the string `'Smith'` to `last_name`. What do you think the following code will assign to the `my_string` variable?

```
full_name = 'Patty Lynn Smith'
my_string = full_name[:]
```

The second statement assigns the entire string `'Patty Lynn Smith'` to `my_string`. The statement is equivalent to:

```
my_string = full_name[0 : len(full_name)]
```

The slicing examples we have seen so far get slices of consecutive characters from strings. Slicing expressions can also have step value, which can cause characters to be skipped in the string. Here is an example of code that uses a slicing expression with a step value:

```
letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
print(letters[0:26:2])
```

The third number inside the brackets is the step value. A step value of 2, as used in this example, causes the slice to contain every second character from the specified range in the string. The code will print the following:

```
ACEGIKMOQSUWY
```

You can also use negative numbers as indexes in slicing expressions to reference positions relative to the end of the string. Here is an example:

```
full_name = 'Patty Lynn Smith'
last_name = full_name[-5:]
```

Recall that Python adds a negative index to the length of a string to get the position referenced by that index. The second statement in this code assigns the string `'Smith'` to the `last_name` variable.

NOTE: Invalid indexes do not cause slicing expressions to raise an exception. For example:

- If the *end* index specifies a position beyond the end of the string, Python will use the length of the string instead.
- If the *start* index specifies a position before the beginning of the string, Python will use 0 instead.
- If the *start* index is greater than the *end* index, the slicing expression will return an empty string.

In the Spotlight:

Extracting Characters from a String

At a university, each student is assigned a system login name, which the student uses to log into the campus computer system. As part of your internship with the university's Information Technology department, you have been asked to write the code that generates system login names for students. You will use the following algorithm to generate a login name:

1. *Get the first three characters of the student's first name. (If the first name is less than three characters in length, use the entire first name.)*
2. *Get the first three characters of the student's last name. (If the last name is less than three characters in length, use the entire last name.)*
3. *Get the last three characters of the student's ID number. (If the ID number is less than three characters in length, use the entire ID number.)*
4. *Concatenate the three sets of characters to generate the login name.*

For example, if a student's name is Amanda Spencer, and her ID number is ENG6721, her login name would be AmaSpe721. You decide to write a function named `get_login_name` that accepts a student's first name, last name, and ID number as arguments and returns the student's login name as a string. You will save the function in a module named `login.py`. This module can then be imported into any Python program that needs to generate a login name. Program 8-3 shows the code for the `login.py` module.

Program 8-3 (login.py)

```
1  # The get_login_name function accepts a first name,
2  # last name, and ID number as arguments. It returns
3  # a system login name.
4
```

```
 5  def get_login_name(first, last, idnumber):
 6      # Get the first three letters of the first name.
 7      # If the name is less than 3 characters, the
 8      # slice will return the entire first name.
 9      set1 = first[0 : 3]
10
11      # Get the first three letters of the last name.
12      # If the name is less than 3 characters, the
13      # slice will return the entire last name.
14      set2 = last[0 : 3]
15
16      # Get the last three characters of the student ID.
17      # If the ID number is less than 3 characters, the
18      # slice will return the entire ID number.
19      set3 = idnumber[-3 :]
20
21      # Put the sets of characters together.
22      login_name = set1 + set2 + set3
23
24      # Return the login name.
25      return login_name
```

The `get_login_name` function accepts three string arguments: a first name, a last name, and an ID number. The statement in line 9 uses a slicing expression to get the first three characters of the string referenced by `first` and assigns those characters, as a string, to the `set1` variable. If the string referenced by `first` is less than three characters long, then the value 3 will be an invalid ending index. If this is the case, Python will use the length of the string as the ending index, and the slicing expression will return the entire string.

The statement in line 14 uses a slicing expression to get the first three characters of the string referenced by `last`, and assigns those characters, as a string, to the `set2` variable. The entire string referenced by `last` will be returned if it is less than three characters.

The statement in line 19 uses a slicing expression to get the last three characters of the string referenced by `idnumber` and assigns those characters, as a string, to the `set3` variable. If the string referenced by `idnumber` is less than three characters, then the value –3 will be an invalid starting index. If this is the case, Python will use 0 as the starting index.

The statement in line 22 assigns the concatenation of `set1`, `set2`, and `set3` to the `login_name` variable. The variable is returned in line 25. Program 8-4 shows a demonstration of the function.

Program 8-4 (generate_login.py)

```
1  # This program gets the user's first name, last name, and
2  # student ID number. Using this data it generates a
3  # system login name.
4
```

(program continues)

Program 8-4 *(continued)*

```
 5  import login
 6
 7  def main():
 8      # Get the user's first name, last name, and ID number.
 9      first = input('Enter your first name: ')
10      last = input('Enter your last name: ')
11      idnumber = input('Enter your student ID number: ')
12
13      # Get the login name.
14      print('Your system login name is:')
15      print(login.get_login_name(first, last, idnumber))
16
17  # Call the main function.
18  main()
```

Program Output (with input shown in bold)

```
Enter your first name: Holly Enter
Enter your last name: Gaddis Enter
Enter your student ID number: CSC34899 Enter
Your system login name is:
HolGad899
```

Program Output (with input shown in bold)

```
Enter your first name: Jo Enter
Enter your last name: Cusimano Enter
Enter your student ID number: BIO4497 Enter
Your system login name is:
JoCus497
```

Checkpoint

8.7 What will the following code display?

```
mystring = 'abcdefg'
print(mystring[2:5])
```

8.8 What will the following code display?

```
mystring = 'abcdefg'
print(mystring[3:])
```

8.9 What will the following code display?

```
mystring = 'abcdefg'
print(mystring[:3])
```

8.10 What will the following code display?

```
mystring = 'abcdefg'
print(mystring[:])
```

8.3 Testing, Searching, and Manipulating Strings

CONCEPT: **Python provides operators and methods for testing strings, searching the contents of strings, and getting modified copies of strings.**

Testing Strings with `in` and `not in`

In Python you can use the `in` operator to determine whether one string is contained in another string. Here is the general format of an expression using the `in` operator with two strings:

```
string1 in string2
```

`string1` and *`string2`* can be either string literals or variables referencing strings. The expression returns true if *`string1`* is found in *`string2`*. For example, look at the following code:

```
text = 'Four score and seven years ago'
if 'seven' in text:
    print('The string "seven" was found.')
else:
    print('The string "seven" was not found.')
```

This code determines whether the string `'Four score and seven years ago'` contains the string `'seven'`. If we run this code it will display:

```
The string "seven" was found.
```

You can use the `not in` operator to determine whether one string is *not* contained in another string. Here is an example:

```
names = 'Bill Joanne Susan Chris Juan Katie'
if 'Pierre' not in names:
    print('Pierre was not found.')
else:
    print('Pierre was found.')
```

If we run this code it will display:

```
Pierre was not found.
```

String Methods

Recall from Chapter 6 that a method is a function that belongs to an object and performs some operation on that object. Strings in Python have numerous methods.[1] In this section we will discuss several string methods for performing the following types of operations:

- Testing the values of strings
- Performing various modifications
- Searching for substrings and replacing sequences of characters

[1] We do not cover all of the string methods in this book. For a comprehensive list of string methods, see the Python documentation at www.python.org.

Here is the general format of a string method call:

```
stringvar.method(arguments)
```

In the general format, *stringvar* is a variable that references a string, *method* is the name of the method that is being called, and *arguments* is one or more arguments being passed to the method. Let's look at some examples.

String Testing Methods

The string methods shown in Table 8-1 test a string for specific characteristics. For example, the `isdigit` method returns true if the string contains only numeric digits. Otherwise, it returns false. Here is an example:

```
string1 = '1200'
if string1.isdigit():
    print(string1, 'contains only digits.')
else:
    print(string1, 'contains characters other than digits.')
```

This code will display

```
1200 contains only digits.
```

Here is another example:

```
string2 = '123abc'
if string2.isdigit():
    print(string2, 'contains only digits.')
else:
    print(string2, 'contains characters other than digits.')
```

This code will display

```
123abc contains characters other than digits.
```

Table 8-1 Some string testing methods

Method	Description
`isalnum()`	Returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise.
`isalpha()`	Returns true if the string contains only alphabetic letters and is at least one character in length. Returns false otherwise.
`isdigit()`	Returns true if the string contains only numeric digits and is at least one character in length. Returns false otherwise.
`islower()`	Returns true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise.
`isspace()`	Returns true if the string contains only whitespace characters and is at least one character in length. Returns false otherwise. (Whitespace characters are spaces, newlines (`\n`), and tabs (`\t`).
`isupper()`	Returns true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise.

Program 8-5 demonstrates several of the string testing methods. It asks the user to enter a string and then displays various messages about the string, depending on the return value of the methods.

Program 8-5 (string_test.py)

```
1  # This program demonstrates several string testing methods.
2
3  def main():
4      # Get a string from the user.
5      user_string = input('Enter a string: ')
6
7      print('This is what I found about that string:')
8
9      # Test the string.
10     if user_string.isalnum():
11        print('The string is alphanumeric.')
12     if user_string.isdigit():
13        print('The string contains only digits.')
14     if user_string.isalpha():
15        print('The string contains only alphabetic characters.')
16     if user_string.isspace():
17        print('The string contains only whitespace characters.')
18     if user_string.islower():
19        print('The letters in the string are all lowercase.')
20     if user_string.isupper():
21        print('The letters in the string are all uppercase.')
22
23 # Call the string.
24 main()
```

Program Output (with input shown in bold)

```
Enter a string: abc Enter
This is what I found about that string:
The string is alphanumeric.
The string contains only alphabetic characters.
The letters in the string are all lowercase.
```

Program Output (with input shown in bold)

```
Enter a string: 123 Enter
This is what I found about that string:
The string is alphanumeric.
The string contains only digits.
```

Program Output (with input shown in bold)

```
Enter a string: 123ABC Enter
This is what I found about that string:
The string is alphanumeric.
The letters in the string are all uppercase.
```

Modification Methods

Although strings are immutable, meaning they cannot be modified, they do have a number of methods that return modified versions of themselves. Table 8-2 lists several of these methods.

Table 8-2 String Modification Methods

Method	Description
`lower()`	Returns a copy of the string with all alphabetic letters converted to lowercase. Any character that is already lowercase, or is not an alphabetic letter, is unchanged.
`lstrip()`	Returns a copy of the string with all leading whitespace characters removed. Leading whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the beginning of the string.
`lstrip(char)`	The *`char`* argument is a string containing a character. Returns a copy of the string with all instances of *`char`* that appear at the beginning of the string removed.
`rstrip()`	Returns a copy of the string with all trailing whitespace characters removed. Trailing whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the end of the string.
`rstrip(char)`	The *`char`* argument is a string containing a character. The method returns a copy of the string with all instances of *`char`* that appear at the end of the string removed.
`strip()`	Returns a copy of the string with all leading and trailing whitespace characters removed.
`strip(char)`	Returns a copy of the string with all instances of *`char`* that appear at the beginning and the end of the string removed.
`upper()`	Returns a copy of the string with all alphabetic letters converted to uppercase. Any character that is already uppercase, or is not an alphabetic letter, is unchanged.

For example, the `lower` method returns a copy of a string with all of its alphabetic letters converted to lowercase. Here is an example:

```
letters = 'WXYZ'
print(letters, letters.lower())
```

This code will print:

```
WXYZ wxyz
```

The `upper` method returns a copy of a string with all of its alphabetic letters converted to uppercase. Here is an example:

```
letters = 'abcd'
print(letters, letters.upper())
```

This code will print:

```
abcd ABCD
```

The `lower` and `upper` methods are useful for making case-insensitive string comparisons. String comparisons are case-sensitive, which means that the uppercase characters are

distinguished from the lowercase characters. For example, in a case-sensitive comparison, the string `'abc'` is not considered the same as the string `'ABC'` or the string `'Abc'` because the case of the characters are different. Sometimes it is more convenient to perform a *case-insensitive* comparison, in which the case of the characters is ignored. In a case-insensitive comparison, the string `'abc'` is considered the same as `'ABC'` and `'Abc'`.

For example, look at the following code:

```
again = 'y'
while again.lower() == 'y':
    print('Hello')
    print('Do you want to see that again?')
    again = input('y = yes, anything else = no: ')
```

Notice that the last statement in the loop asks the user to enter `y` to see the message displayed again. The loop iterates as long as the expression `again.lower() == 'y'` is true. The expression will be true if the `again` variable references either `'y'` or `'Y'`.

Similar results can be achieved by using the `upper` method, as shown here:

```
again = 'y'
while again.upper() == 'Y':
    print('Hello')
    print('Do you want to see that again?')
    again = input('y = yes, anything else = no: ')
```

Searching and Replacing

Programs commonly need to search for substrings, or strings that appear within other strings. For example, suppose you have a document opened in your word processor, and you need to search for a word that appears somewhere in it. The word that you are searching for is a substring that appears inside a larger string, the document.

Table 8-3 lists some of the Python string methods that search for substrings, as well as a method that replaces the occurrences of a substring with another string.

Table 8-3 Search and replace methods

Method	Description
`endswith(substring)`	The *substring* argument is a string. The method returns true if the string ends with *substring*.
`find(substring)`	The *substring* argument is a string. The method returns the lowest index in the string where *substring* is found. If *substring* is not found, the method returns –1.
`replace(old, new)`	The *old* and *new* arguments are both strings. The method returns a copy of the string with all instances of *old* replaced by *new*.
`startswith(substring)`	The *substring* argument is a string. The method returns true if the string starts with *substring*.

The `endswith` method determines whether a string ends with a specified substring. Here is an example:

```
filename = input('Enter the filename: ')
if filename.endswith('.txt'):
    print('That is the name of a text file.')
elif filename.endswith('.py'):
    print('That is the name of a Python source file.')
elif filename.endswith('.doc'):
    print('That is the name of a word processing document.')
else:
    print('Unknown file type.')
```

The `startswith` method works like the `endswith` method, but determines whether a string begins with a specified substring.

The `find` method searches for a specified substring within a string. The method returns the lowest index of the substring, if it is found. If the substring is not found, the method returns –1. Here is an example:

```
string = 'Four score and seven years ago'
position = string.find('seven')
if position != -1:
    print('The word "seven" was found at index', position)
else:
    print('The word "seven" was not found.')
```

This code will display:

```
The word "seven" was found at index 15
```

The `replace` method returns a copy of a string, where every occurrence of a specified substring has been replaced with another string. For example, look at the following code:

```
string = 'Four score and seven years ago'
new_string = string.replace('years', 'days')
print(new_string)
```

This code will display:

```
Four score and seven days ago
```

In the Spotlight:

Validating the Characters in a Password

At the university, passwords for the campus computer system must meet the following requirements:

- The password must be at least seven characters long.
- It must contain at least one uppercase letter.

- It must contain at least one lowercase letter.
- It must contain at least one numeric digit.

When a student sets up his or her password, the password must be validated to ensure it meets these requirements. You have been asked to write the code that performs this validation. You decide to write a function named `valid_password` that accepts the password as an argument and returns either true or false, to indicate whether it is valid. Here is the algorithm for the function, in pseudocode:

```
valid_password function:
  Set the correct_length variable to false
  Set the has_uppercase variable to false
  Set the has_lowercase variable to false
  Set the has_digit variable to false
  If the password's length is seven characters or greater:
    Set the correct_length variable to true
    for each character in the password:
      if the character is an uppercase letter:
        Set the has_uppercase variable to true
      if the character is a lowercase letter:
        Set the has_lowercase variable to true
      if the character is a digit:
        Set the has_digit variable to true

  If correct_length and has_uppercase and has_lowercase and has_digit:
    Set the is_valid variable to true
  else:
    Set the is_valid variable to false

    Return the is_valid variable
```

Earlier (in the previous In the Spotlight section) you created a function named `get_login_name` and stored that function in the `login` module. Because the `valid_password` function's purpose is related to the task of creating a student's login account, you decide to store the `valid_password` function in the `login` module as well. Program 8-6 shows the login module with the `valid_password` function added to it. The function begins at line 34.

Program 8-6 (login.py)

```
 1 # The get_login_name function accepts a first name,
 2 # last name, and ID number as arguments. It returns
 3 # a system login name.
 4
 5 def get_login_name(first, last, idnumber):
 6     # Get the first three letters of the first name.
 7     # If the name is less than 3 characters, the
 8     # slice will return the entire first name.
 9     set1 = first[0 : 3]
10
```

(program continues)

Program 8-6 *(continued)*

```
11      # Get the first three letters of the last name.
12      # If the name is less than 3 characters, the
13      # slice will return the entire last name.
14      set2 = last[0 : 3]
15
16      # Get the last three characters of the student ID.
17      # If the ID number is less than 3 characters, the
18      # slice will return the entire ID number.
19      set3 = idnumber[-3 :]
20
21      # Put the sets of characters together.
22      login_name = set1 + set2 + set3
23
24      # Return the login name.
25      return login_name
26
27  # The valid_password function accepts a password as
28  # an argument and returns either true or false to
29  # indicate whether the password is valid. A valid
30  # password must be at least 7 characters in length,
31  # have at least one uppercase letter, one lowercase
32  # letter, and one digit.
33
34  def valid_password(password):
35      # Set the Boolean variables to false.
36      correct_length = False
37      has_uppercase = False
38      has_lowercase = False
39      has_digit = False
40
41      # Begin the validation. Start by testing the
42      # password's length.
43      if len(password) >= 7:
44          correct_length = True
45
46          # Test each character and set the
47          # appropriate flag when a required
48          # character is found.
49          for ch in password:
50              if ch.isupper():
51                  has_uppercase = True
52              if ch.islower():
53                  has_lowercase = True
54              if ch.isdigit():
55                  has_digit = True
```

```
56
57      # Determine whether all of the requirements
58      # are met. If they are, set is_valid to true.
59      # Otherwise, set is_valid to false.
60      if correct_length and has_uppercase and \
61          has_lowercase and has_digit:
62          is_valid = True
63      else:
64          is_valid = False
65
66      # Return the is_valid variable.
67      return is_valid
```

Program 8-7 imports the login module and demonstrates the `valid_password` function.

Program 8-7 (validate_password.py)

```
1  # This program gets a password from the user and
2  # validates it.
3
4  import login
5
6  def main():
7      # Get a password from the user.
8      password = input('Enter your password: ')
9
10     # Validate the password.
11     while not login.valid_password(password):
12         print('That password is not valid.')
13         password = input('Enter your password: ')
14
15     print('That is a valid password.')
16
17 # Call the main function.
18 main()
```

Program Output (with input shown in bold)

```
Enter your password: bozo Enter
That password is not valid.
Enter your password: kangaroo Enter
That password is not valid.
Enter your password: Tiger9 Enter
That password is not valid.
Enter your password: Leopard6 Enter
That is a valid password.
```

The Repetition Operator

In Chapter 7 you learned how to duplicate a list with the repetition operator (*). The repetition operator works with strings as well. Here is the general format:

```
string_to_copy * n
```

The repetition operator creates a string that contains *n* repeated copies of *string_to_copy*. Here is an example:

```
my_string = 'w' * 5
```

After this statement executes, `my_string` will reference the string `'wwwww'`. Here is another example:

```
print('Hello' * 5)
```

This statement will print:

```
HelloHelloHelloHelloHello
```

Program 8-8 demonstrates the repetition operator.

Program 8-8 (repetition_operator.py)

```
1  # This program demonstrates the repetition operator.
2
3  def main():
4      # Print nine rows increasing in length.
5      for count in range(1, 10):
6          print('Z' * count)
7
8      # Print nine rows decreasing in length.
9      for count in range(8, 0, -1):
10         print('Z' * count)
11
12 # Call the main function.
13 main()
```

Program Output

```
Z
ZZ
ZZZ
ZZZZ
ZZZZZ
ZZZZZZ
ZZZZZZZ
ZZZZZZZZ
ZZZZZZZZZ
ZZZZZZZZ
ZZZZZZZ
```

```
ZZZZZZ
ZZZZZ
ZZZZ
ZZZ
ZZ
Z
```

Splitting a String

Strings in Python have a method named `split` that returns a list containing the words in the string. Program 8-9 shows an example.

Program 8-9 (string_split.py)

```
1  # This program demonstrates the split method.
2
3  def main():
4      # Create a string with multiple words.
5      my_string = 'One two three four'
6
7      # Split the string.
8      word_list = my_string.split()
9
10     # Print the list of words.
11     print(word_list)
12
13 # Call the main function.
14 main()
```

Program Output

```
['One', 'two', 'three', 'four']
```

By default, the split method uses spaces as separators (that is, it returns a list of the words in the string that are separated by spaces). You can specify a different separator by passing it as an argument to the split method. For example, suppose a string contains a date, as shown here:

```
date_string = '11/26/2014'
```

If you want to break out the month, day, and year as items in a list, you can call the `split` method using the `'/'` character as a separator, as shown here:

```
date_list = date_string.split('/')
```

After this statement executes, the `date_list` variable will reference this list:

```
['11', '26', '2014']
```

Program 8-10 demonstrates this.

Program 8-10 (split_date.py)

```
1  # This program calls the split method, using the
2  # '/' character as a separator.
3
4  def main():
5      # Create a string with a date.
6      date_string = '11/26/2014'
7
8      # Split the date.
9      date_list = date_string.split('/')
10
11     # Display each piece of the date.
12     print('Month:', date_list[0])
13     print('Day:', date_list[1])
14     print('Year:', date_list[2])
15
16 # Call the main function.
17 main()
```

Program Output

```
Month: 11
Day: 26
Year: 2014
```

Checkpoint

8.11 Write code using the `in` operator that determines whether `'d'` is in `mystring`.

8.12 Assume the variable `big` references a string. Write a statement that converts the string it references to lowercase and assigns the converted string to the variable `little`.

8.13 Write an `if` statement that displays "Digit" if the string referenced by the variable `ch` contains a numeric digit. Otherwise, it should display "No digit."

8.14 What is the output of the following code?

```
ch = 'a'
ch2 = ch.upper()
print(ch, ch2)
```

8.15 Write a loop that asks the user "Do you want to repeat the program or quit? (R/Q)". The loop should repeat until the user has entered an R or Q (either uppercase or lowercase).

8.16 What will the following code display?

```
var = '$'
print(var.upper())
```