Chapter 4. Lists

One more topic you'll need to understand before you can begin writing programs in earnest is the list data type and its cousin, the tuple. Lists and tuples can contain multiple values, which makes it easier to write programs that handle large amounts of data. And since lists themselves can contain other lists, you can use them to arrange data into hierarchical structures.

In this chapter, I'll discuss the basics of lists. I'll also teach you about methods, which are functions that are tied to values of a certain data type. Then I'll briefly cover the list-like tuple and string data types and how they compare to list values. In the next chapter, I'll introduce you to the dictionary data type.

The List Data Type

A *list* is a value that contains multiple values in an ordered sequence. The term *list value* refers to the list itself (which is a value that can be stored in a variable or passed to a function like any other value), not the values inside the list value. A list value looks like this: `['cat', 'bat', 'rat', 'elephant']`. Just as string values are typed with quote characters to mark where the string begins and ends, a list begins with an opening square bracket and ends with a closing square bracket, `[]`. Values inside the list are also called *items*. Items are separated with commas (that is, they are *comma-delimited*). For example, enter the following into the interactive shell:

```
>>> [1, 2, 3]

[1, 2, 3]

>>> ['cat', 'bat', 'rat', 'elephant']

['cat', 'bat', 'rat', 'elephant']

>>> ['hello', 3.1415, True, None, 42]

['hello', 3.1415, True, None, 42]

❶ >>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> spam

['cat', 'bat', 'rat', 'elephant']
```

The spamvariable ❶ is still assigned only one value: the list value. But the list value itself contains other values. The value []is an empty list that contains no values, similar to '', the empty string.

Getting Individual Values in a List with Indexes

Say you have the list `['cat', 'bat', 'rat', 'elephant']` stored in a variable named `spam`. The Python code `spam[0]` would evaluate to `'cat'`, and `spam[1]` would evaluate to `'bat'`, and so on. The integer inside the square brackets that follows the list is called an *index*. The first value in the list is at index `0`, the second value is at index `1`, the third value is at index `2`, and so on. Figure 4-1 shows a list value assigned to `spam`, along with what the index expressions would evaluate to.

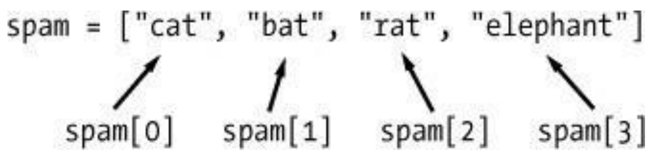


Figure 4-1. A list value stored in the variable `spam`, showing which value each index refers to

For example, type the following expressions into the interactive shell. Start by assigning a list to the variable `spam`.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> spam[0]

'cat'

>>> spam[1]
```

```
   'bat'
   >>> spam[2]
   'rat'
   >>> spam[3]
   'elephant'
   >>> ['cat', 'bat', 'rat', 'elephant'][3]
   'elephant'
➊ >>> 'Hello ' + spam[0]
➋ 'Hello cat'
   >>> 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
   'The bat ate the cat.'
```

Notice that the expression 'Hello ' + spam[0]➊ evaluates to 'Hello ' + 'cat'

because spam[0]evaluates to the string 'cat'. This expression in turn evaluates to the string value `'Hello cat'` ➋.

Python will give you an `IndexError` error message if you use an index that exceeds the number of values in your list value.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[10000]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    spam[10000]
IndexError: list index out of range
```

Indexes can be only integer values, not floats. The following example will cause a `TypeError` error:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1]
'bat'
>>> spam[1.0]
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    spam[1.0]
TypeError: list indices must be integers, not float
>>> spam[int(1.0)]
'bat'
```

Lists can also contain other list values. The values in these lists of lists can be accessed using multiple indexes, like so:

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
```

```
'bat'

>>> spam[1][4]

50
```

The first index dictates which list value to use, and the second indicates the value within the list value. For example, `spam[0][1]` prints `'bat'`, the second value in the first list. If you only use one index, the program will print the full list value at that index.

Negative Indexes

While indexes start at `0` and go up, you can also use negative integers for the index. The integer value `-1` refers to the last index in a list, the value `-2` refers to the second-to-last index in a list, and so on. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> spam[-1]

'elephant'

>>> spam[-3]

'bat'

>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'

'The elephant is afraid of the bat.'
```

Getting Sublists with Slices

Just as an index can get a single value from a list, a *slice* can get several values from a list, in the form of a new list. A slice is typed between square brackets, like an index, but it has two integers separated by a colon. Notice the difference between indexes and slices.

- `spam[2]` is a list with an index (one integer).
- `spam[1:4]` is a list with a slice (two integers).

In a slice, the first integer is the index where the slice starts. The second integer is the

index where the slice ends. A slice goes up to, but will not include, the value at the second index. A slice evaluates to a new list value. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> spam[0:4]

['cat', 'bat', 'rat', 'elephant']

>>> spam[1:3]

['bat', 'rat']

>>> spam[0:-1]

['cat', 'bat', 'rat']
```

As a shortcut, you can leave out one or both of the indexes on either side of the colon in the slice. Leaving out the first index is the same as using `0`, or the beginning of the list. Leaving out the second index is the same as using the length of the list, which will slice to the end of the list. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> spam[:2]

['cat', 'bat']

>>> spam[1:]
```

```
['bat', 'rat', 'elephant']

>>> spam[:]

['cat', 'bat', 'rat', 'elephant']
```

Getting a List's Length with len()

The `len()` function will return the number of values that are in a list value passed to it, just like it can count the number of characters in a string value. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'moose']

>>> len(spam)

3
```

Changing Values in a List with Indexes

Normally a variable name goes on the left side of an assignment statement, like `spam = 42`. However, you can also use an index of a list to change the value at that index. For example, `spam[1] = 'aardvark'` means "Assign the value at index `1` in the list `spam` to the string `'aardvark'`." Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> spam[1] = 'aardvark'

>>> spam

['cat', 'aardvark', 'rat', 'elephant']

>>> spam[2] = spam[1]

>>> spam

['cat', 'aardvark', 'aardvark', 'elephant']

>>> spam[-1] = 12345

>>> spam

['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

The `+` operator can combine two lists to create a new list value in the same way it combines two strings into a new string value. The `*` operator can also be used with a list and an integer value to replicate the list. Enter the following into the interactive shell:

```
>>> [1, 2, 3] + ['A', 'B', 'C']

[1, 2, 3, 'A', 'B', 'C']

>>> ['X', 'Y', 'Z'] * 3

['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']

>>> spam = [1, 2, 3]

>>> spam = spam + ['A', 'B', 'C']

>>> spam

[1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with del Statements

The `del` statement will delete values at an index in a list. All of the values in the list after the deleted value will be moved up one index. For example, enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> del spam[2]

>>> spam

['cat', 'bat', 'elephant']

>>> del spam[2]

>>> spam

['cat', 'bat']
```

The `del` statement can also be used on a simple variable to delete it, as if it were an “unassignment” statement. If you try to use the variable after deleting it, you will get a `NameError` error because the variable no longer exists.

In practice, you almost never need to delete simple variables. The `del` statement is mostly used to delete values from lists.

Working with Lists

When you first begin writing programs, it’s tempting to create many individual variables to store a group of similar values. For example, if I wanted to store the names of my cats, I might be tempted to write code like this:

```
catName1 = 'Zophie'
catName2 = 'Pooka'
catName3 = 'Simon'
catName4 = 'Lady Macbeth'
catName5 = 'Fat-tail'
catName6 = 'Miss Cleo'
```

(I don’t actually own this many cats, I swear.) It turns out that this is a bad way to write code. For one thing, if the number of cats changes, your program will never be able to store more cats than you have variables. These types of programs also have a lot of duplicate or nearly identical code in them. Consider how much duplicate code is in the following program, which you should enter into the file editor and save as *allMyCats1.py*:

```
print('Enter the name of cat 1:')
catName1 = input()

print('Enter the name of cat 2:')
catName2 = input()

print('Enter the name of cat 3:')
catName3 = input()

print('Enter the name of cat 4:')
catName4 = input()

print('Enter the name of cat 5:')
catName5 = input()

print('Enter the name of cat 6:')
catName6 = input()

print('The cat names are:')

print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' + catName4 + ' ' +
catName5 + ' ' + catName6)
```

Instead of using multiple, repetitive variables, you can use a single variable that contains a list value. For example, here’s a new and improved version of the *allMyCats1.py* program. This new version uses a single list and can store any number of cats that the user types in. In a new file editor window, type the following source code and save it as *allMyCats2.py*:

```
catNames = []
while True:

    print('Enter the name of cat ' + str(len(catNames) + 1) +
      ' (Or enter nothing to stop.):')

    name = input()
    if name == '':

        break

    catNames = catNames + [name] # list concatenation
print('The cat names are:')

for name in catNames:
    print('  ' + name)
```

When you run this program, the output will look something like this:

```
Enter the name of cat 1 (Or enter nothing to stop.):

Zophie

Enter the name of cat 2 (Or enter nothing to stop.):

Pooka

Enter the name of cat 3 (Or enter nothing to stop.):

Simon

Enter the name of cat 4 (Or enter nothing to stop.):

Lady Macbeth

Enter the name of cat 5 (Or enter nothing to stop.):

Fat-tail

Enter the name of cat 6 (Or enter nothing to stop.):

Miss Cleo

Enter the name of cat 7 (Or enter nothing to stop.):

The cat names are:


  Zophie
  Pooka
  Simon

  Lady Macbeth
  Fat-tail
  Miss Cleo
```

The benefit of using a list is that your data is now in a structure, so your program is much more flexible in processing the data than it would be with several repetitive variables.

Using for Loops with Lists

In Chapter 2, you learned about using `for` loops to execute a block of code a certain number of times. Technically, a `for` loop repeats the code block once for each value in a list or list-like value. For example, if you ran this code:

```
for i in range(4):
    print(i)
```

the output of this program would be as follows:

```
0

1

2
```

```
3
```

This is because the return value from `range(4)` is a list-like value that Python considers similar to `[0, 1, 2, 3]`. The following program has the same output as the previous one:

```
for i in [0, 1,
   2, 3]:
   print(i)
```

What the previous `for` loop actually does is loop through its clause with the variable `i` set to a successive value in the `[0, 1, 2, 3]` list in each iteration.

NOTE

In this book, I use the term list-like *to refer to data types that are technically named* sequences. *You don't need to know the technical definitions of this term, though*.

A common Python technique is to use `range(len(someList))` with a `for` loop to iterate over the indexes of a list. For example, enter the following into the interactive shell:

```
>>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']

>>> for i in range(len(supplies)):

    print('Index ' + str(i) + ' in supplies is: ' + supplies[i])


Index 0 in supplies is: pens
Index 1 in supplies is: staplers

Index 2 in supplies is: flame-throwers
Index 3 in supplies is: binders
```

Using `range(len(supplies))` in the previously shown `for` loop is handy because the code in the loop can access the index (as the variable `i`) and the value at that index (as `supplies[i]`). Best of all, `range(len(supplies))` will iterate through all the indexes of `supplies`, no matter how many items it contains.

The in and not in Operators

You can determine whether a value is or isn't in a list with the `in` and `not in` operators. Like other operators, `in` and `not in` are used in expressions and connect two values: a value to look for in a list and the list where it may be found. These expressions will evaluate to a Boolean value. Enter the following into the interactive shell:

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']

True

>>> spam = ['hello', 'hi', 'howdy', 'heyas']

>>> 'cat' in spam

False

>>> 'howdy' not in spam

False

>>> 'cat' not in spam

True
```

For example, the following program lets the user type in a pet name and then checks to see whether the name is in a list of pets. Open a new file editor window, enter the following code, and save it as *myPets.py*:

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')

name = input()
```

```
if name not in myPets:

    print('I do not have a pet named ' + name)
else:

    print(name + ' is my pet.')
```

The output may look something like this:

```
Enter a pet name:

Footfoot

I do not have a pet named Footfoot
```

The Multiple Assignment Trick

The *multiple assignment trick* is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So instead of doing this:

```
>>> cat = ['fat', 'black', 'loud']

>>> size = cat[0]

>>> color = cat[1]

>>> disposition = cat[2]
```

you could type this line of code:

```
>>> cat = ['fat', 'black', 'loud']

>>> size, color, disposition = cat
```

The number of variables and the length of the list must be exactly equal, or Python will give you a `ValueError`:

```
>>> cat = ['fat', 'black', 'loud']

>>> size, color, disposition, name = cat

Traceback (most recent call last):

  File "<pyshell#84>", line 1, in <module>
    size, color, disposition, name = cat

ValueError: need more than 3 values to unpack
```

Augmented Assignment Operators

When assigning a value to a variable, you will frequently use the variable itself. For example, after assigning `42` to the variable `spam`, you would increase the value in `spam` by `1` with the following code:

```
>>> spam = 42

>>> spam = spam + 1

>>> spam

43
```

As a shortcut, you can use the augmented assignment operator `+=` to do the same thing:

```
>>> spam = 42

>>> spam += 1

>>> spam

43
```

There are augmented assignment operators for the `+`, `-`, `*`, `/`, and `%` operators, described in Table 4-1.

Table 4-1. The Augmented Assignment Operators

Augmented assignment statement	Equivalent assignment statement
`spam = spam + 1`	`spam += 1`
`spam = spam - 1`	`spam -= 1`
`spam = spam * 1`	`spam *= 1`
`spam = spam / 1`	`spam /= 1`
`spam = spam % 1`	`spam %= 1`

The +=operator can also do string and list concatenation, and the *=operator can do string and list replication. Enter the following into the interactive shell:

```
>>> spam = 'Hello'

>>> spam += ' world!'

>>> spam

'Hello world!'

>>> bacon = ['Zophie']

>>> bacon *= 3

>>> bacon

['Zophie', 'Zophie', 'Zophie']
```

Methods

A *method* is the same thing as a function, except it is "called on" a value. For example, if a list value were stored in spam, you would call the index() list method (which I'll explain next) on that list like so: spam.index('hello'). The method part comes after the value, separated by a period.

Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list.

Finding a Value in a List with the index() Method

List values have an `index()` method that can be passed a value, and if that value exists in the list, the index of the value is returned. If the value isn't in the list, then Python produces a `ValueError` error. Enter the following into the interactive shell:

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']

>>> spam.index('hello')

0

>>> spam.index('heyas')

3

>>> spam.index('howdy howdy howdy')

Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    spam.index('howdy howdy howdy')

ValueError: 'howdy howdy howdy' is not in list
```

When there are duplicates of the value in the list, the index of its first appearance is returned. Enter the following into the interactive shell, and notice that index()returns 1, not 3:

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']

>>> spam.index('Pooka')

1
```

Adding Values to Lists with the append() and insert() Methods

To add new values to a list, use the `append()` and `insert()` methods. Enter the following into the interactive shell to call the `append()` method on a list value stored in the variable `spam`:

```
>>> spam = ['cat', 'dog', 'bat']

>>> spam.append('moose')

>>> spam

['cat', 'dog', 'bat', 'moose']
```

The previous `append()` method call adds the argument to the end of the list. The `insert()` method can insert a value at any index in the list. The first argument to `insert()` is the index for the new value, and the second argument is the new value to be inserted. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'bat']

>>> spam.insert(1, 'chicken')

>>> spam

['cat', 'chicken', 'dog', 'bat']
```

Notice that the code is spam.append('moose') and spam.insert(1, 'chicken'), not spam = spam.append('moose') and spam = spam.insert(1, 'chicken'). Neither append()nor insert()gives the new value of spamas its return value. (In fact, the return value of append()and insert()is None, so you definitely wouldn't want to store this as the new variable value.) Rather, the list is modified *in place*. Modifying a list in place is

covered in more detail later in Mutable and Immutable Data Types.

Methods belong to a single data type. The append() and insert() methods are list methods and can be called only on list values, not on other values such as strings or integers. Enter the following into the interactive shell, and note the AttributeErrorerror messages that show up:

```
>>> eggs = 'hello'

>>> eggs.append('world')

Traceback (most recent call last):

  File "<pyshell#19>", line 1, in <module>
    eggs.append('world')

AttributeError: 'str' object has no attribute 'append'

>>> bacon = 42

>>> bacon.insert(1, 'world')

Traceback (most recent call last):

  File "<pyshell#22>", line 1, in <module>
    bacon.insert(1, 'world')

AttributeError: 'int' object has no attribute 'insert'
```

Removing Values from Lists with remove()

The `remove()` method is passed the value to be removed from the list it is called on. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> spam.remove('bat')

>>> spam

['cat', 'rat', 'elephant']
```

Attempting to delete a value that does not exist in the list will result in a `ValueError` error. For example, enter the following into the interactive shell and notice the error that is displayed:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> spam.remove('chicken')

Traceback (most recent call last):

  File "<pyshell#11>", line 1, in <module>
    spam.remove('chicken')

ValueError: list.remove(x): x not in list
```

If the value appears multiple times in the list, only the first instance of the value will be removed. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'cat', 'hat', 'cat']

>>> spam.remove('cat')

>>> spam

['bat', 'rat', 'cat', 'hat', 'cat']
```

The `del` statement is good to use when you know the index of the value you want to remove from the list. The `remove()` method is good when you know the value you want to remove from the list.

Sorting the Values in a List with the sort() Method

Lists of number values or lists of strings can be sorted with the `sort()` method. For example, enter the following into the interactive shell:

```
>>> spam = [2, 5, 3.14, 1, -7]

>>> spam.sort()

>>> spam

[-7, 1, 2, 3.14, 5]

>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']

>>> spam.sort()

>>> spam

['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

You can also pass `True` for the `reverse` keyword argument to have `sort()` sort the values in reverse order. Enter the following into the interactive shell:

```
>>> spam.sort(reverse=True)

>>> spam

['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

There are three things you should note about the `sort()` method. First, the `sort()` method sorts the list in place; don't try to capture the return value by writing code like `spam = spam.sort()`.

Second, you cannot sort lists that have both number values *and* string values in them, since Python doesn't know how to compare these values. Type the following into the interactive shell and notice the `TypeError` error:

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']

>>> spam.sort()

Traceback (most recent call last):

 File "<pyshell#70>", line 1, in <module>
   spam.sort()

TypeError: unorderable types: str() < int()
```

Third, `sort()` uses "ASCIIbetical order" rather than actual alphabetical order for sorting strings. This means uppercase letters come before lowercase letters. Therefore, the lowercase *a* is sorted so that it comes *after* the uppercase *Z*. For an example, enter the following into the interactive shell:

```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']

>>> spam.sort()

>>> spam

['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

If you need to sort the values in regular alphabetical order, pass `str. lower` for the `key` keyword argument in the `sort()` method call.

```
>>> spam = ['a', 'z', 'A', 'Z']

>>> spam.sort(key=str.lower)

>>> spam

['a', 'A', 'z', 'Z']
```

This causes the `sort()` function to treat all the items in the list as if they were lowercase without actually changing the values in the list.

Example Program: Magic 8 Ball with a List

Using lists, you can write a much more elegant version of the previous chapter's Magic 8 Ball program. Instead of several lines of nearly identical `elif` statements, you can create a single list that the code works with. Open a new file editor window and enter the following code. Save it as *magic8Ball2.py*.

```
import random


messages = ['It is certain',
    'It is decidedly so',
    'Yes  definitely',
    'Reply hazy try again',
    'Ask again later',

    'Concentrate and ask again',
    'My reply is no',
    'Outlook not so good',
    'Very doubtful']

    print(messages[random.r
    andint(0, len(messages)
    - 1)])
```

EXCEPTIONS TO INDENTATION RULES IN PYTHON

In most cases, the amount of indentation for a line of code tells Python what block it is in. There are some exceptions to this rule, however. For example, lists can actually span several lines in the source code file. The indentation of these lines do not matter; Python knows that until it sees the ending square bracket, the list is not finished. For example, you can have code that looks like this:

```
spam = ['apples',
    'oranges',

                    'bananas',

'cats']
print(spam)
```

Of course, practically speaking, most people use Python's behavior to make their lists look pretty and readable, like the messages list in the Magic 8 Ball program.

You can also split up a single instruction across multiple lines using the \ line continuation character at the end. Think of \ as saying, "This instruction continues on the next line." The indentation on the line after a \ line continuation is not significant. For example, the following is valid Python code:

When you run this program, you'll see that it works the same as the previous *magic8Ball.py* program.

Notice the expression you use as the index into messages: random.randint(0, len(messages) - 1). This produces a random number to use for the index, regardless of the size of messages. That is, you'll get a random number between 0 and the value of len(messages) - 1. The benefit of this approach is that you can easily add and remove strings to the messageslist without changing other lines of code. If you later update your code, there will be fewer lines you have to change and fewer chances for you to introduce bugs.

List-like Types: Strings and Tuples

Lists aren't the only data types that represent ordered sequences of values. For example, strings and lists are actually similar, if you consider a string to be a "list" of single text characters. Many of the things you can do with lists can also be done with strings: indexing; slicing; and using them with `for` loops, with `len()`, and with the `in` and `not in` operators. To see this, enter the following into the interactive shell:

```
>>> name = 'Zophie'
>>> name[0]
'Z'
>>> name[-2]
'i'
>>> name[0:4]
'Zoph'
>>> 'Zo' in name
True
>>> 'z' in name
False
>>> 'p' not in name
False
>>> for i in name:
        print('* * * ' + i + ' * * *')

* * * Z * * *

* * * o * * *

* * * p * * *

* * * h * * *

* * * i * * *
* * * e * * *
```

Mutable and Immutable Data Types

But lists and strings are different in an important way. A list value is a *mutable* data type: It can have values added, removed, or changed. However, a string is *immutable*: It cannot be changed. Trying to reassign a single character in a string results in a `TypeError` error, as you can see by entering the following into the interactive shell:

```
>>> name = 'Zophie a cat'

>>> name[7] = 'the'

Traceback (most recent call last):

  File "<pyshell#50>", line 1, in <module>
    name[7] = 'the'

TypeError: 'str' object does not support item assignment
```

The proper way to "mutate" a string is to use slicing and concatenation to build a *new* string by copying from parts of the old string. Enter the following into the interactive shell:

```
>>> name = 'Zophie a cat'

>>> newName = name[0:7] + 'the' + name[8:12]

>>> name

'Zophie a cat'

>>> newName

'Zophie the cat'
```

We used `[0:7]` and `[8:12]` to refer to the characters that we don't wish to replace. Notice that the original `'Zophie a cat'` string is not modified because strings are immutable.

Although a list value *is* mutable, the second line in the following code does not modify the list `eggs`:

```
>>> eggs = [1, 2, 3]

>>> eggs = [4, 5, 6]

>>> eggs

[4, 5, 6]
```

The list value in `eggs` isn't being changed here; rather, an entirely new and different list value (`[4, 5, 6]`) is overwriting the old list value (`[1, 2, 3]`). This is depicted in Figure 4-2.

If you wanted to actually modify the original list in `eggs` to contain `[4, 5, 6]`, you would have to do something like this:

```
>>> eggs = [1, 2, 3]

>>> del eggs[2]

>>> del eggs[1]

>>> del eggs[0]

>>> eggs.append(4)

>>> eggs.append(5)

>>> eggs.append(6)

>>> eggs

[4, 5, 6]
```

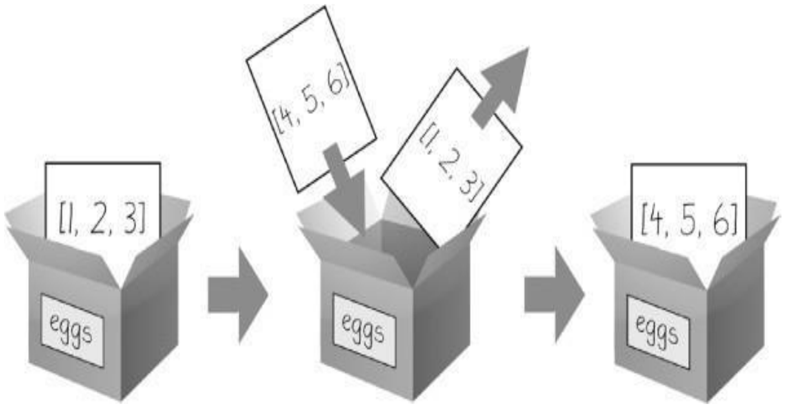


Figure 4-2. When `eggs = [4, 5, 6]` *is executed, the contents of* `eggs` *are replaced with a new list value.*

In the first example, the list value that `eggs` ends up with is the same list value it started with. It's just that this list has been changed, rather than overwritten. Figure 4-3 depicts the seven changes made by the first seven lines in the previous interactive shell example.

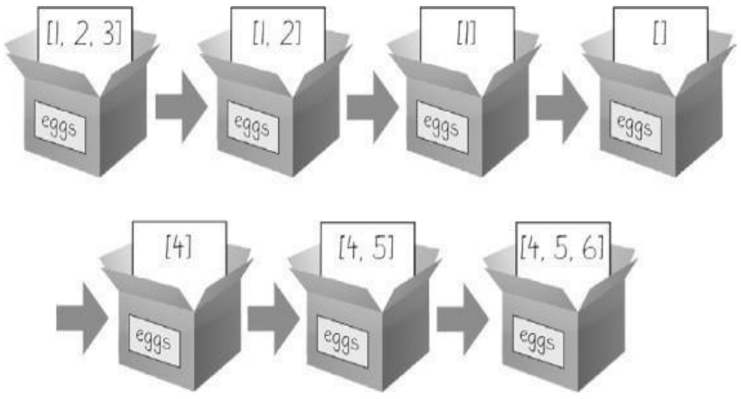


Figure 4-3. The `del` *statement and the* `append()` *method modify the same list value in place.*

Changing a value of a mutable data type (like what the del statement and append() method do in the previous example) changes the value in place, since the variable's value is not replaced with a new list value.

Mutable versus immutable types may seem like a meaningless distinction, but Passing References will explain the different behavior when calling functions with mutable arguments versus immutable arguments. But first, let's find out about the tuple data type,which is an immutable form of the list data type.

The Tuple Data Type

The *tuple* data type is almost identical to the list data type, except in two ways. First, tuples are typed with parentheses, `(` and `)`, instead of square brackets, `[` and `]`. For example, enter the following into the interactive shell:

```
>>> eggs = ('hello', 42, 0.5)

>>> eggs[0]

'hello'

>>> eggs[1:3]

(42, 0.5)

>>> len(eggs)

3
```

But the main way that tuples are different from lists is that tuples, like strings, are immutable. Tuples cannot have their values modified, appended, or removed. Enter the following into the interactive shell, and look at the TypeErrorerror message:

```
>>> eggs = ('hello', 42, 0.5)

>>> eggs[1] = 99

Traceback (most recent call last):

  File "<pyshell#5>", line 1, in <module>
    eggs[1] = 99

TypeError: 'tuple' object does not support item assignment
```

If you have only one value in your tuple, you can indicate this by placing a trailing comma after the value inside the parentheses. Otherwise, Python will think you've just typed a value inside regular parentheses. The comma is what lets Python know this is a tuple value. (Unlike some other programming languages, in Python it's fine to have a trailing comma after the last item in a list or tuple.) Enter the following `type()` function calls into the interactive shell to see the distinction:

```
>>> type(('hello',))

<class 'tuple'>

>>> type(('hello'))

<class 'str'>
```

You can use tuples to convey to anyone reading your code that you don't intend for that sequence of values to change. If you need an ordered sequence of values that never changes, use a tuple. A second benefit of using tuples instead of lists is that, because they are immutable and their contents don't change, Python can implement some optimizations that make code using tuples slightly faster than code using lists.

Converting Types with the list() and tuple() Functions

Just like how `str(42)` will return `'42'`, the string representation of the integer `42`, the functions `list()` and `tuple()` will return list and tuple versions of the values passed to them. Enter the following into the interactive shell, and notice that the return value is of a different data type than the value passed:

```
>>> tuple(['cat', 'dog', 5])

('cat', 'dog', 5)

>>> list(('cat', 'dog', 5))

['cat', 'dog', 5]

>>> list('hello')

['h', 'e', 'l', 'l', 'o']
```

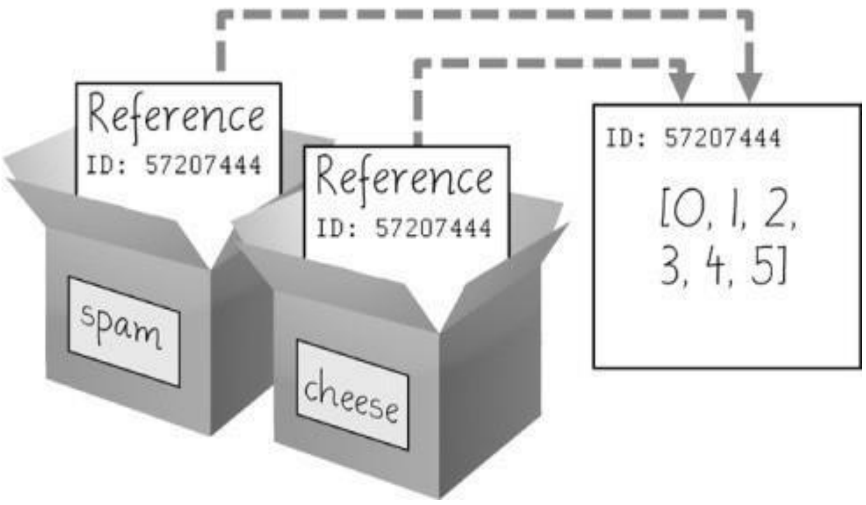
Converting a tuple to a list is handy if you need a mutable version of a tuple value.

References

As you've seen, variables store strings and integer values. Enter the following into the interactive shell:

```
>>> spam = 42

>>> cheese = spam

>>> spam = 100

>>> spam

100

>>> cheese

42
```

You assign `42` to the `spam` variable, and then you copy the value in `spam` and assign it to the variable `cheese`. When you later change the value in `spam` to `100`, this doesn't affect the value in `cheese`.



This is because `spam` and `cheese` are different variables that store different values.

But lists don't work this way. When you assign a list to a variable, you are actually assigning a list *reference* to the variable. A reference is a value that points to some bit of data, and a list reference is a value that points to a list. Here is some code that will make this distinction easier to understand. Enter this into the interactive shell:

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]

❷ >>> cheese = spam

❸ >>> cheese[1] = 'Hello!'

   >>> spam

   [0, 'Hello!', 2, 3, 4, 5]

   >>> cheese

   [0, 'Hello!', 2, 3, 4, 5]
```

This might look odd to you. The code changed only the cheeselist, but it seems that both the cheeseand spamlists have changed.

When you create the list ❶, you assign a reference to it in the `spam` variable. But the next line ❷ copies only the list reference in `spam` to `cheese`, not the list value itself. This means the values stored in `spam` and `cheese` now both refer to the same list. There is only one underlying list because the list itself was never actually copied. So when you modify the first element of `cheese` ❸, you are modifying the same list that `spam` refers to.

Remember that variables are like boxes that contain values. The previous figures in this chapter show that lists in boxes aren't exactly accurate because list variables don't actually contain lists — they contain *references* to lists. (These references will have ID numbers that Python uses internally, but you can ignore them.) Using boxes as a metaphor for variables, Figure 4-4 shows what happens when a list is assigned to the `spam` variable.

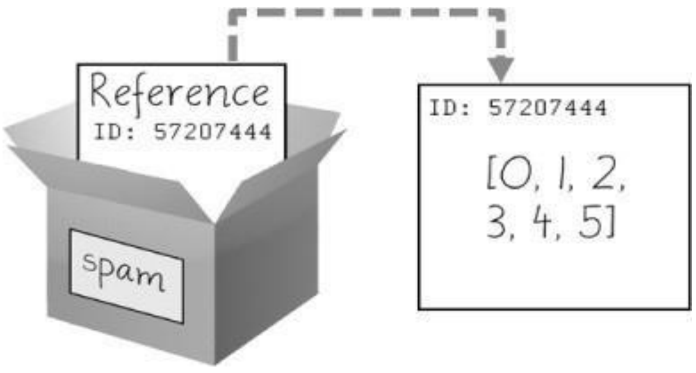


Figure 4-4. `spam = [0, 1, 2, 3, 4, 5]` *stores a reference to a list, not the actual list.*

Then, in Figure 4-5, the reference in `spam` is copied to `cheese`. Only a new reference was created and stored in `cheese`, not a new list. Note how both references refer to the same list.

Figure 4-5. `spam = cheese` *copies the reference, not the list.*

When you alter the list that `cheese` refers to, the list that `spam` refers to is also changed, because both `cheese` and `spam` refer to the same list. You can see this in Figure 4-6.

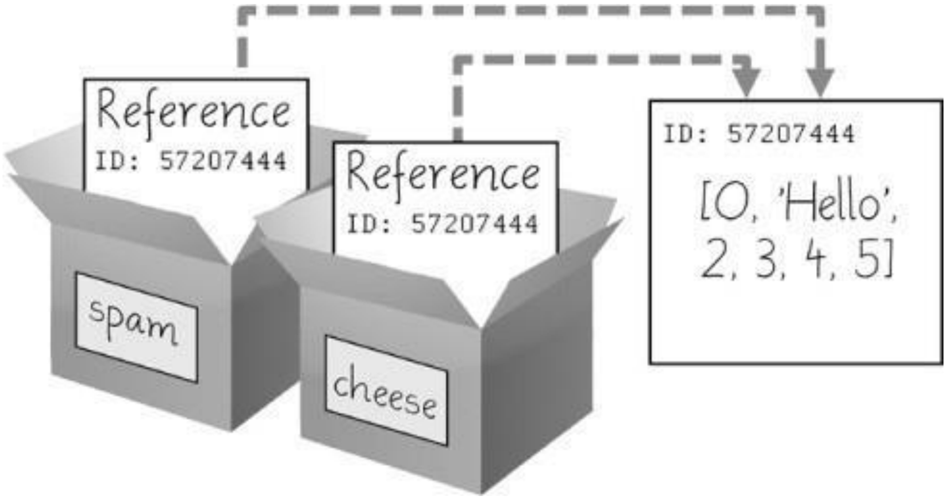


Figure 4-6. `cheese[1] = 'Hello!'` *modifies the list that both variables refer to.*

Variables will contain references to list values rather than list values themselves. But for strings and integer values, variables simply contain the string or integer value. Python uses references whenever variables must store values of mutable data types, such as lists ordictionaries. For values of immutable data types such as strings, integers, or tuples, Python variables will store the value itself.

Although Python variables technically contain references to list or dictionary values, people often casually say that the variable contains the list or dictionary.

Passing References

References are particularly important for understanding how arguments get passed to functions. When a function is called, the values of the arguments are copied to the parameter variables. For lists (and dictionaries, which I'll describe in the next chapter), this means a copy of the reference is used for the parameter. To see the consequences of this, open a new file editor window, enter the following code, and save it as *passingReference.py*:

```
def eggs(someParameter):
    someParameter.append('Hello')


spam = [1, 2, 3]
eggs(spam)
print(spam)
```

Notice that when `eggs()` is called, a return value is not used to assign a new value to `spam`. Instead, it modifies the list in place, directly. When run, this program produces the following output:

```
[1, 2, 3, 'Hello']
```

Even though `spam` and `someParameter` contain separate references, they both refer to the same list. This is why the `append('Hello')` method call inside the function affects the list even after the function call has returned.

Keep this behavior in mind: Forgetting that Python handles list and dictionary variables this way can lead to confusing bugs.

The copy Module's copy() and deepcopy() Functions

Although passing around references is often the handiest way to deal with lists and dictionaries, if the function modifies the list or dictionary that is passed, you may not want these changes in the original list or dictionary value. For this, Python provides a module named copy that provides both the `copy()` and deepcopy() functions. The first of these, copy.copy(), can be used to make a duplicate copy of a mutable value like a list or dictionary, not just a copy of a reference. Enter the following into the interactive shell:

```
>>> import copy

>>> spam = ['A', 'B', 'C', 'D']

>>> cheese = copy.copy(spam)

>>> cheese[1] = 42

>>> spam

['A', 'B', 'C', 'D']

>>> cheese

['A', 42, 'C', 'D']
```

Now the spam and cheese variables refer to separate lists, which is why only the list in cheese is modified when you assign 42 at index 7. As you can see in Figure 4-7, the reference ID numbers are no longer the same for both variables because the variables refer to independent lists.

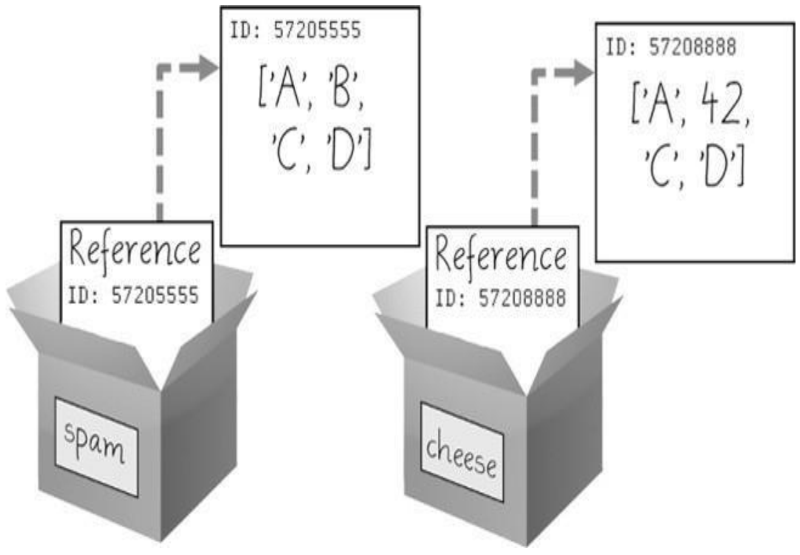


Figure 4-7. cheese = copy.copy(spam) *creates a second list that can be modified independently of the first.*

If the list you need to copy contains lists, then use the copy.deepcopy() function instead of copy.copy(). The deepcopy() function will copy these inner lists as well.

Chapter 5. Dictionaries and Structuring Data

In this chapter, I will cover the dictionary data type, which provides a flexible way to access and organize data. Then, combining dictionaries with your knowledge of lists from the previous chapter, you'll learn how to create a data structure to model a tic-tac-toe board.

The Dictionary Data Type

Like a list, a *dictionary* is a collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called *keys*, and a key with its associated value is called a *key-value pair*.

In code, a dictionary is typed with braces, `{}`. Enter the following into the interactive shell:

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

This assigns a dictionary to the `myCat` variable. This dictionary's keys are `'size'`, `'color'`, and `'disposition'`. The values for these keys are `'fat'`, `'gray'`, and `'loud'`, respectively. You can access these values through their keys:

```
>>> myCat['size']

'fat'

>>> 'My cat has ' + myCat['color'] + ' fur.'

'My cat has gray fur.'
```

Dictionaries can still use integer values as keys, just like lists use integers for indexes, but they do not have to start at `0` and can be any number.

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}
```

Dictionaries vs. Lists

Unlike lists, items in dictionaries are unordered. The first item in a list named spamwould be spam[0]. But there is no "first" item in a dictionary. While the order of items matters for determining whether two lists are the same, it does not matter in what order the key- value pairs are typed in a dictionary. Enter the following into the interactive shell:

```
>>> spam = ['cats', 'dogs', 'moose']

>>> bacon = ['dogs', 'moose', 'cats']

>>> spam == bacon

False

>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}

>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}

>>> eggs == ham

True
```

Because dictionaries are not ordered, they can't be sliced like lists.

Trying to access a key that does not exist in a dictionary will result in a `KeyError` error message, much like a list's "out-of-range" `IndexError` error message. Enter the following into the interactive shell, and notice the error message that shows up because there is no `'color'` key:

```
>>> spam = {'name': 'Zophie', 'age': 7}

>>> spam['color']

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    spam['color']

KeyError: 'color'
```

Though dictionaries are not ordered, the fact that you can have arbitrary values for the keys allows you to organize your data in powerful ways. Say you wanted your program to store data about your friends' birthdays. You can use a dictionary with the names as keys and the birthdays as values. Open a new file editor window and enter the following code. Save it as *birthdays.py*.

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

   while True:

       print('Enter a name: (blank to quit)')
       name = input()

       if name == '':
           break

❷      if name in birthdays:

❸          print(birthdays[name] + ' is the birthday of ' + name)
       else:

           print('I do not have birthday information for ' + name)
           print('What is their birthday?')

           bday = input()

❹          birthdays[name] = bday
           print('Birthday database updated.')
```

You create an initial dictionary and store it in `birthdays` ❶. You can see if the entered name exists as a key in the dictionary with the `in` keyword ❷, just as you did for lists. If the name is in the dictionary, you access the associated value using square brackets ❸; if not, you can add it using the same square bracket syntax combined with the assignment operator ❹.

When you run this program, it will look like this:

```
Enter a name: (blank to quit)

Alice

Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve

I do not have birthday information for Eve
What is their birthday?

Dec 5

Birthday database updated.
Enter a name: (blank to quit)
Eve

Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

Of course, all the data you enter in this program is forgotten when the program terminates.

The keys(), values(), and items() Methods

There are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: keys(), values(), and items(). The values returned by these methods are not true lists: They cannot be modified and do not have an append() method. But these data types (dict_keys, dict_values, and dict_items, respectively) *can* be used in for loops. To see how these methods work, enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
        print(v)


red
42
```

Here, a `for` loop iterates over each of the values in the `spam` dictionary. A `for` loop can also iterate over the keys or both keys and values:

```
>>> for k in spam.keys():

        print(spam[k])


color
age

>>> for i in spam.items():

        print(i)


('color', 'red')('age', 42)
```

Using the `keys()`, `values()`, and `items()` methods, a `for` loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively. Notice that the values in the `dict_items` value returned by the `items()` method are tuples of the key and value.

If you want a true list from one of these methods, pass its list-like return value to the `list()` function. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}

>>> spam.keys()

dict_keys(['color', 'age'])

>>> list(spam.keys())

['color', 'age']
```

The `list(spam.keys())` line takes the `dict_keys` value returned from `keys()` and passes it to `list()`, which then returns a list value of `['color', 'age']`.

You can also use the multiple assignment trick in a `for` loop to assign the key and value to separate variables. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}

>>> for k, v in spam.items():

        print('Key: ' + k + ' Value: ' + str(v))


Key: age Value: 42
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

Recall from the previous chapter that the `in` and `not in` operators can check whether a value exists in a list. You can also use these operators to see whether a certain key or value exists in a dictionary. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
```

```
False
```

In the previous example, notice that 'color' in spam is essentially a shorter version of writing 'color' in spam.keys(). This is always the case: If you ever want to check whether a value is (or isn't) a key in the dictionary, you can simply use the in(or not in) keyword with the dictionary value itself.

The get() Method

It's tedious to check whether a key exists in a dictionary before accessing that key's value. Fortunately, dictionaries have a get() method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

Enter the following into the interactive shell:

```
>>> picnicItems = {'apples': 5, 'cups': 2}

>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'

'I am bringing 2 cups.'

>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'

'I am bringing 0 eggs.'
```

Because there is no `'eggs'` key in the `picnicItems` dictionary, the default value `0` is returned by the `get()` method. Without using `get()`, the code would have caused an error message, such as in the following example:

```
>>> picnicItems = {'apples': 5, 'cups': 2}

>>> 'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'

Traceback (most recent call last):

  File "<pyshell#34>", line 1, in <module>

    'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
KeyError: 'eggs'
```

The setdefault() Method

You'll often have to set a value in a dictionary for a certain key only if that key does not already have a value. The code looks something like this:

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:

    spam['color'] = 'black'
```

The `setdefault()` method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist. If the key does exist, the `setdefault()` method returns the key's value. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Pooka', 'age': 5}

>>> spam.setdefault('color', 'black')

'black'

>>> spam

{'color': 'black', 'age': 5, 'name': 'Pooka'}

>>> spam.setdefault('color', 'white')

'black'

>>> spam

{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

The first time `setdefault()` is called, the dictionary in `spam` changes to `{'color': 'black', 'age': 5, 'name': 'Pooka'}`. The method returns the value `'black'` because this is now the value set for the key `'color'`. When `spam.setdefault('color', 'white')` is called next, the value for that key is *not* changed to `'white'` because `spam` already has a key named `'color'`.

The `setdefault()` method is a nice shortcut to ensure that a key exists. Here is a short program that counts the number of occurrences of each letter in a string. Open the file editor window and enter the following code, saving it as *characterCount.py*:

```
message = 'It was a bright cold day in April, and the clocks were striking thirteen.'
count = {}


for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1


print(count)
```

The program loops over each character in the `message` variable's string, counting how often each character appears. The `setdefault()` method call ensures that the key is in the `count` dictionary (with a default value of `0`) so the program doesn't throw a `KeyError` error when `count[character] = count[character] + 1` is executed. When you run this program, the output will look like this:

```
{' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2,  'i':

6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y':  1}
```

From the output, you can see that the lowercase letter *c* appears 3 times, the space

character appears 13 times, and the uppercase letter *A* appears 1 time. This program will work no matter what string is inside the `message` variable, even if the string is millions of characters long!

Pretty Printing

If you import the pprint module into your programs, you'll have access to the pprint() and pformat()functions that will "pretty print" a dictionary's values. This is helpful when you want a cleaner display of the items in a dictionary than what print()provides.
Modify the previous *characterCount.py* program and save it as *prettyCharacterCount.py*.

```
import pprint

message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'

count = {}


for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1


pprint.pprint(count)
```

This time, when the program is run, the output looks much cleaner, with the keys sorted.

```
{' ': 13,

 ',': 1,

 '.': 1,

 'A': 1,

 'I': 1,

 'a': 4,

 'b': 1,
    'c': 3,

    'd': 3,

    'e': 5,

    'g': 2,

    'h': 3,

    'i': 6,

    'k': 2,

    'l': 3,

    'n': 4,

    'o': 2,

    'p': 1,

    'r': 5,

    's': 3,

    't': 6,

    'w': 2,

    'y': 1}
```

The `pprint.pprint()` function is especially helpful when the dictionary itself contains nested lists or dictionaries.

If you want to obtain the prettified text as a string value instead of displaying it on the screen, call `pprint.pformat()` instead. These two lines are equivalent to each other:

```
pprint.pprint(someDictionaryValue)
print(pprint.pformat(someDictionaryValue))
```

Using Data Structures to Model Real-World Things

Even before the Internet, it was possible to play a game of chess with someone on the other side of the world. Each player would set up a chessboard at their home and then take turns mailing a postcard to each other describing each move. To do this, the players needed a way to unambiguously describe the state of the board and their moves.

In *algebraic chess notation*, the spaces on the chessboard are identified by a number and letter coordinate, as in Figure 5-1.

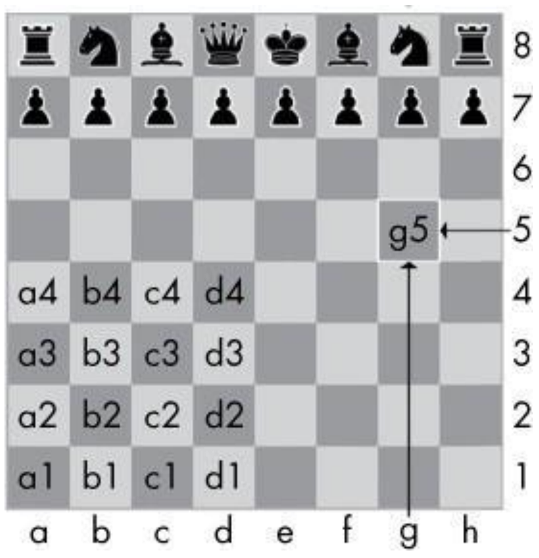


Figure 5-1. The coordinates of a chessboard in algebraic chess notation

The chess pieces are identified by letters: *K* for king, *Q* for queen, *R* for rook, *B* for bishop, and *N* for knight. Describing a move uses the letter of the piece and the coordinates of its destination. A pair of these moves describes what happens in a single turn (with white going first); for instance, the notation *2. Nf3 Nc6* indicates that white moved a knight to f3 and black moved a knight to c6 on the second turn of the game.

There's a bit more to algebraic notation than this, but the point is that you can use it to unambiguously describe a game of chess without needing to be in front of a chessboard. Your opponent can even be on the other side of the world! In fact, you don't even need a physical chess set if you have a good memory: You can just read the mailed chess moves and update boards you have in your imagination.

Computers have good memories. A program on a modern computer can easily store billions of strings like `'2. Nf3 Nc6'`. This is how computers can play chess without having a physical chessboard. They model data to represent a chessboard, and you can write code to work with this model.

This is where lists and dictionaries can come in. You can use them to model real-world things, like chessboards. For the first example, you'll use a game that's a little simpler than chess: tic-tac-toe.

A Tic-Tac-Toe Board

A tic-tac-toe board looks like a large hash symbol (#) with nine slots that can each contain an *X*, an *O*, or a blank. To represent the board with a dictionary, you can assign each slot a string-value key, as shown in Figure 5-2.

You can use string values to represent what's in each slot on the board: `'X'`, `'O'`, or `' '` (a space character). Thus, you'll need to store nine strings. You can use a dictionary of values for this. The string value with the key `'top-R'` can represent the top-right corner, the string value with the key `'low-L'` can represent the bottom-left corner, the string value with the key `'mid-M'` can represent the middle, and so on.

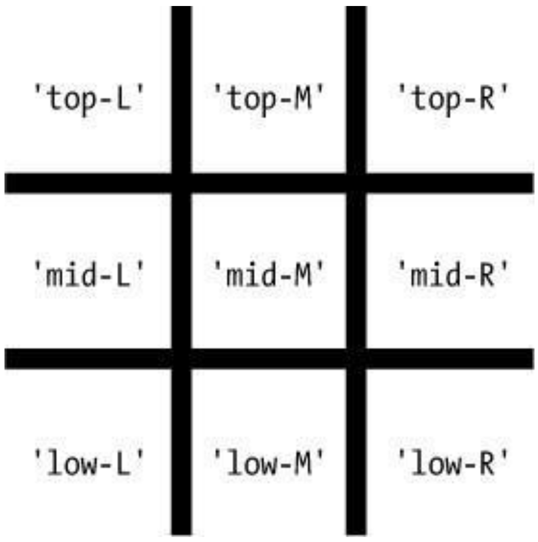


Figure 5-2. The slots of a tic-tactoe board with their corresponding keys

This dictionary is a data structure that represents a tic-tac-toe board. Store this board-as-a- dictionary in a variable named `theBoard`. Open a new file editor window, and enter the following source code, saving it as *ticTacToe.py*:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',

            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',

            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

The data structure stored in the `theBoard` variable represents the tic-tactoe board in Figure 5-3.

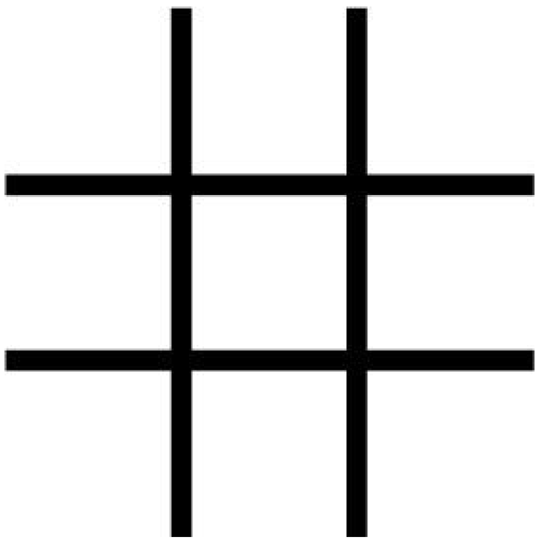


Figure 5-3. An empty tic-tac-toe board

Since the value for every key in `theBoard` is a single-space string, this dictionary represents a completely clear board. If player X went first and chose the middle space, you could represent that board with this dictionary:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',

            'mid-L': ' ', 'mid-M': 'X', 'mid-R': ' ',

            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

The data structure in `theBoard` now represents the tic-tac-toe board in Figure 5-4.

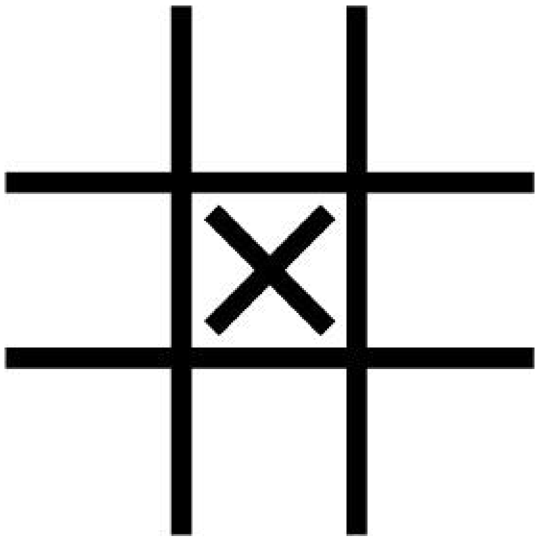


Figure 5-4. The first move

A board where player O has won by placing *O*s across the top might look like this:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',

            'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',

            'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}
```

The data structure in `theBoard` now represents the tic-tac-toe board in Figure 5-5.

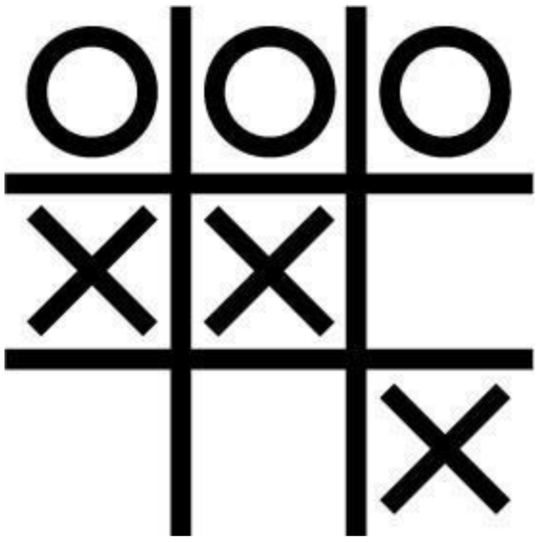


Figure 5-5. Player O wins.

Of course, the player sees only what is printed to the screen, not the contents of variables. Let's create a function to print the board dictionary onto the screen. Make the following addition to *ticTacToe.py* (new code is in bold):

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',

            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',

            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):

    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')

    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

When you run this program, `printBoard()` will print out a blank tic-tactoe board.

```
 | 
 
-+-+-
 
 | 
 
-+-+-
 
 | 
```

The `printBoard()` function can handle any tic-tac-toe data structure you pass it. Try changing the code to the following:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O', 'mid-L': 'X', 'mid-M':

'X', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}


def printBoard(board):

    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')

    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')

    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

Now when you run this program, the new board will be printed to the screen.

```
O|O|O

-+-+-

X|X|

-+-+-

 | |X
```

Because you created a data structure to represent a tic-tac-toe board and wrote code in `printBoard()` to interpret that data structure, you now have a program that "models" the tic-tac-toe board. You could have organized your data structure differently (for example, using keys like `'TOP-LEFT'` instead of `'top-L'`), but as long as the code works with your data structures, you will have a correctly working program.

For example, the `printBoard()` function expects the tic-tac-toe data structure to be a dictionary with keys for all nine slots. If the dictionary you passed was missing, say, the `'mid-L'` key, your program would no longer work.

```
O|O|O

-+-+-

Traceback (most recent call last):

  File "ticTacToe.py", line 10, in <module>
    printBoard(theBoard)

  File "ticTacToe.py", line 6, in printBoard

    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
KeyError: 'mid-L'
```

Now let's add code that allows the players to enter their moves. Modify the *ticTacToe.py* program to look like this:

```
    theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ', 'mid-L': ' ', 'mid-M': '

    ', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': ' '}


    def printBoard(board):

        print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
        print('-+-+-')

        print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
        print('-+-+-')

        print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])

    turn = 'X'

    for i in range(9):

❶        printBoard(theBoard)

         print('Turn for ' + turn + '. Move on which space?')

❷        move = input()

❸        theBoard[move] = turn

❹        if turn == 'X':
             turn = 'O'

         else:

             turn = 'X'

    printBoard(theBoard)
```

The new code prints out the board at the start of each new turn ❶, gets the active player'smove ❷, updates the game board accordingly ❸, and then swaps the active player ❹
before moving on to the next turn.

When you run this program, it will look something like this:

```
 | |

-+-+-

 | |

-+-+-

 | |

Turn for X. Move on which space?

mid-M

 | |

-+-+-

 |X|

-+-+-

 | |

Turn for O. Move on which space?

low-L

 | |

-+-+-
```

```
  |X|
-+-+-
O| |

--snip--
O|O|X
-+-+-
X|X|O
-+-+-
O| |X
Turn for X. Move on which space?
low-M
O|O|X
-+-+-
X|X|O
-+-+-
O|X|X
```

This isn't a complete tic-tac-toe game — for instance, it doesn't ever check whether a player has won — but it's enough to see how data structures can be used in programs.

NOTE

If you are curious, the source code for a complete tic-tac-toe program is described in the resources available from

Nested Dictionaries and Lists

Modeling a tic-tac-toe board was fairly simple: The board needed only a single dictionary value with nine key-value pairs. As you model more complicated things, you may find you need dictionaries and lists that contain other dictionaries and lists. Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values. For example, here's a program that uses a dictionary that contains other dictionaries in order to see who is bringing what to a picnic. The `totalBrought()` function can read this data structure and calculate the total number of an item being brought by all the guests.

```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
             'Bob': {'ham sandwiches': 3, 'apples': 2},
             'Carol': {'cups': 3, 'apple pies': 1}}

def totalBrought(guests, item):
    numBrought = 0
❶   for k, v in guests.items():
❷       numBrought = numBrought + v.get(item, 0)
```

```
        return numBrought

    print('Number of things being brought:')

    print(' - Apples         ' + str(totalBrought(allGuests, 'apples')))
    print(' - Cups           ' + str(totalBrought(allGuests, 'cups')))
    print(' - Cakes          ' + str(totalBrought(allGuests, 'cakes')))

    print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches')))
    print(' - Apple Pies     ' + str(totalBrought(allGuests, 'apple pies')))
```

Inside the `totalBrought()` function, the `for` loop iterates over the key-value pairs in `guests` ❶. Inside the loop, the string of the guest's name is assigned to `k`, and the dictionary of picnic items they're bringing is assigned to `v`. If the item parameter exists as a key in this dictionary, it's value (the quantity) is added to `numBrought` ❷. If it does not exist as a key, the `get()` method returns `0` to be added to `numBrought`.

The output of this program looks like this:

```
Number of things being brought:

- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies     1
```

This may seem like such a simple thing to model that you wouldn't need to bother with writing a program to do it. But realize that this same `totalBrought()` function could easily handle a dictionary that contains thousands of guests, each bringing *thousands* of different picnic items. Then having this information in a data structure along with the `totalBrought()` function would save you a lot of time!

You can model things with data structures in whatever way you like, as long as the rest of the code in your program can work with the data model correctly. When you first begin programming, don't worry so much about the "right" way to model data. As you gain more experience, you may come up with more efficient models, but the important thing is that the data model works for your program's needs.

Chapter 6. Manipulating Strings

Text is one of the most common forms of data your programs will handle. You already know how to concatenate two string values together with the + operator, but you can do much more than that. You can extract partial strings from string values, add or remove spacing, convert letters to lowercase or uppercase, and check that strings are formatted correctly. You can even write Python code to access the clipboard for copying and pasting text.

In this chapter, you'll learn all this and more. Then you'll work through two different programming projects: a simple password manager and a program to automate the boring chore of formatting pieces of text.

Working with Strings

Let's look at some of the ways Python lets you write, print, and access strings in your code.

String Literals

Typing string values in Python code is fairly straightforward: They begin and end with a single quote. But then how can you use a quote inside a string? Typing **`'That is Alice's cat.'`** won't work, because Python thinks the string ends after `Alice`, and the rest (`s cat.'`) is invalid Python code. Fortunately, there are multiple ways to type strings.

Double Quotes

Strings can begin and end with double quotes, just as they do with single quotes. One benefit of using double quotes is that the string can have a single quote character in it. Enter the following into the interactive shell:

```
>>> spam = "That is Alice's cat."
```

Since the string begins with a double quote, Python knows that the single quote is part of the string and not marking the end of the string. However, if you need to use both single quotes and double quotes in the string, you'll need to use escape characters.

Escape Characters

An *escape character* lets you use characters that are otherwise impossible to put into a string. An escape character consists of a backslash (\) followed by the character you want to add to the string. (Despite consisting of two characters, it is commonly referred to as a singular escape character.) For example, the escape character for a single quote is `\'`. You can use this inside a string that begins and ends with single quotes. To see how escape characters work, enter the following into the interactive shell:

```
>>> spam = 'Say hi to Bob\'s mother.'
```

Python knows that since the single quote in `Bob\'s` has a backslash, it is not a single quote meant to end the string value. The escape characters `\'` and `\"` let you put single quotes and double quotes inside your strings, respectively.

Table 6-1 lists the escape characters you can use.

Table 6-1. Escape Characters

Escape character	Prints as
`\'`	Single quote
`\"`	Double

`\t`	Tab
`\n`	Newline (linebreak)
`\\`	Backslash

Enter the following into the interactive shell:

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")

Hello there!
How are you?
I'm doing fine.
```

Raw Strings

You can place an `r` before the beginning quotation mark of a string to make it a raw string. A *raw string* completely ignores all escape characters and prints any backslash that appears in the string. For example, type the following into the interactive shell:

```
>>> print(r'That is Carol\'s cat.')

That is Carol\'s cat.
```

Because this is a raw string, Python considers the backslash as part of the string and not as the start of an escape character. Raw strings are helpful if you are typing string values that contain many backslashes, such as the strings used for regular expressions described in the next chapter.

Multiline Strings with TripleQuotes

While you can use the `\n` escape character to put a newline into a string, it is often easier to use multiline strings. A multiline string in Python begins and ends with either three single quotes or three double quotes. Any quotes, tabs, or newlines in between the "triple quotes" are considered part of the string. Python's indentation rules for blocks do not apply to lines inside a multiline string.

Open the file editor and write the following:

```
print('''Dear Alice,


Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,

Bob''')
```

Save this program as *catnapping.py* and run it. The output will look like this:

```
Dear Alice,


Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,

Bob
```

Notice that the single quote character in `Eve's` does not need to be escaped. Escaping single and double quotes is optional in raw strings. The following `print()` call would print identical text but doesn't use a multiline string:

```
print('Dear Alice,\n\nEve\'s cat has been arrested for catnapping, cat
burglary, and extortion.\n\nSincerely,\nBob')
```

Multiline Comments

While the hash character (#) marks the beginning of a comment for the rest of the line, a multiline string is often used for comments that span multiple lines. The following is perfectly valid Python code:

```
"""This is a test Python program.

Written by Al Sweigart al@inventwithpython.com


This program was designed for Python 3, not Python 2.
"""
```

```
def spam():

    """This is a multiline comment to help
    explain what the spam() function does."""
    print('Hello!')
```

Indexing and Slicing Strings

Strings use indexes and slices the same way lists do. You can think of the string `'Hello world!'` as a list and each character in the string as an item with a corresponding index.

```
'   H   e   l   l   o       w   o   r   l   d    !   '
    0   1   2   3   4   5   6   7   8   9   10   11
```

The space and exclamation point are included in the character count, so `'Hello world!'` is 12 characters long, from `H` at index 0 to `!` at index 11. Enter the

following into the interactive shell:

```
>>> spam = 'Hello world!'

>>> spam[0]

'H'

>>> spam[4]

'o'

>>> spam[-1]

'!'

>>> spam[0:5]

'Hello'

>>> spam[:5]

'Hello'

>>> spam[6:]

'world!'
```

If you specify an index, you'll get the character at that position in the string. If you specify a range from one index to another, the starting index is included and the ending index is not. That's why, if spam is `'Hello world!'`, `spam[0:5]` is `'Hello'`. The substring you get from `spam[0:5]` will include everything from `spam[0]` to `spam[4]`, leaving out the space at index 5.

Note that slicing a string does not modify the original string. You can capture a slice from one variable in a separate variable. Try typing the following into the interactive shell:

```
>>> spam = 'Hello world!'

>>> fizz = spam[0:5]

>>> fizz

'Hello'
```

By slicing and storing the resulting substring in another variable, you can have both the whole string and the substring handy for quick, easy access.

The in and not in Operators with Strings

The `in` and `not in` operators can be used with strings just like with list values. An expression with two strings joined using `in` or `not in` will evaluate to a Boolean `True` or `False`. Enter the following into the interactive shell:

```
>>> 'Hello' in 'Hello World'
```

```
True

>>> 'Hello' in 'Hello'

True

>>> 'HELLO' in 'Hello World'

False

>>> '' in 'spam'

True

>>> 'cats' not in 'cats and dogs'

False
```

These expressions test whether the first string (the exact string, case sensitive) can be found within the second string.

Useful String Methods

Several string methods analyze strings or create transformed string values. This section describes the methods you'll be using most often.

The upper(), lower(), isupper(), and islower() String Methods

The `upper()` and `lower()` string methods return a new string where all the letters in the original string have been converted to uppercase or lower-case, respectively. Nonletter characters in the string remain unchanged. Enter the following into the interactive shell:

```
>>> spam = 'Hello world!'

>>> spam = spam.upper()

>>> spam

'HELLO WORLD!'

>>> spam = spam.lower()

>>> spam

'hello world!'
```

Note that these methods do not change the string itself but return new string values. If you want to change the original string, you have to call `upper()` or `lower()` on the string and then assign the new string to the variable where the original was stored. This is why you must use `spam = spam.upper()` to change the string in `spam` instead of simply `spam.upper()`. (This is just like if a variable `eggs` contains the value `10`. Writing `eggs + 3` does not change the value of `eggs`, but `eggs = eggs + 3` does.)

The `upper()` and `lower()` methods are helpful if you need to make a case-insensitive comparison. The strings `'great'` and `'GREat'` are not equal to each other. But in the following small program, it does not matter whether the user types `Great`, `GREAT`, or `grEAT`, because the string is first converted to lowercase.

```
print('How are you?')
feeling = input()

if feeling.lower() == 'great':
    print('I feel great too.')

else:

    print('I hope the rest of your day is good.')
```

When you run this program, the question is displayed, and entering a variation on `great`, such as `GREat`, will still give the output `I feel great too`. Adding code to your program to handle variations or mistakes in user input, such as inconsistent capitalization, will make your programs easier to use and less likely to fail.

```
How are you?

GREat

I feel great too.
```

The `isupper()` and `islower()` methods will return a Boolean `True` value if the string has at least one letter and all the letters are uppercase or lowercase, respectively. Otherwise, the method returns `False`. Enter the following into the interactive shell, and notice what each method call returns:

```
>>> spam = 'Hello world!'

>>> spam.islower()

False

>>> spam.isupper()

False

>>> 'HELLO'.isupper()

True

>>> 'abc12345'.islower()

True

>>> '12345'.islower()

False

>>> '12345'.isupper()

False
```

Since the `upper()` and `lower()` string methods themselves return strings, you can call string methods on *those* returned string values as well. Expressions that do this will look like a chain of method calls. Enter the following into the interactive shell:

```
>>> 'Hello'.upper()

'HELLO'

>>> 'Hello'.upper().lower()

'hello'

>>> 'Hello'.upper().lower().upper()

'HELLO'

>>> 'HELLO'.lower()

'hello'

>>> 'HELLO'.lower().islower()

True
```

The isX String Methods

Along with `islower()` and `isupper()`, there are several string methods that have names beginning with the word *is*. These methods return a Boolean value that describes the nature of the string. Here are some common `is`*X* string methods:

- `isalpha()` returns `True` if the string consists only of letters and is not blank. `isalnum()` returns
- `True` if the string consists only of letters and numbers and is not blank.
- `isdecimal()` returns `True` if the string consists only of numeric characters and is not blank.
- `isspace()` returns `True` if the string consists only of spaces, tabs, and new-lines and is not blank.
- `istitle()` returns `True` if the string consists only of words that begin with an uppercase letter

followed by only lowercase letters.

Enter the following into the interactive shell:

```
>>> 'hello'.isalpha()

True

>>> 'hello123'.isalpha()

False

>>> 'hello123'.isalnum()

True

>>> 'hello'.isalnum()

True

>>> '123'.isdecimal()

True

>>> '    '.isspace()

True

>>> 'This Is Title Case'.istitle()

True

>>> 'This Is Title Case 123'.istitle()

True

>>> 'This Is not Title Case'.istitle()

False

>>> 'This Is NOT Title Case Either'.istitle()

False
```

The `isX` string methods are helpful when you need to validate user input. For example, the following program repeatedly asks users for their age and a password until they provide valid input. Open a new file editor window and enter this program, saving it as *validateInput.py*:

```
while True:

    print('Enter your age:')

    age = input()

    if age.isdecimal():
        break

    print('Please enter a number for your age.')


while True:

    print('Select a new password (letters and numbers only):')
    password = input()

    if password.isalnum():
        break

    print('Passwords can only have letters and numbers.')
```

In the first `while` loop, we ask the user for their age and store their input in `age`. If `age` is a valid (decimal) value, we break out of this first `while` loop and move on to the second, which asks for a password. Otherwise, we inform the user that they need to enter a number and again ask them to enter their age. In the second `while` loop, we ask for a password, store the user's input in `password`, and break out of the loop if the input was alphanumeric. If it wasn't, we're not satisfied so we tell the user the password needs to be alphanumeric and again ask them to enter a password.

When run, the program's output looks like this:

```
Enter your age:

forty two

Please enter a number for your age.
Enter your age:

42

Select a new password (letters and numbers only):

secr3t!

Passwords can only have letters and numbers.
Select a new password (letters and numbers only):

secr3t
```

Calling `isdecimal()` and `isalnum()` on variables, we're able to test whether the values stored in those variables are decimal or not, alphanumeric or not. Here, these tests help us reject the input `forty two` and accept `42`, and reject `secr3t!` and accept `secr3t`.

The startswith() and endswith() String Methods

The `startswith()` and `endswith()` methods return `True` if the string value they are called on begins or ends (respectively) with the string passed to the method; otherwise, they return `False`. Enter the following into the interactive shell:

```
>>> 'Hello world!'.startswith('Hello')

True

>>> 'Hello world!'.endswith('world!')

True

>>> 'abc123'.startswith('abcdef')

False

>>> 'abc123'.endswith('12')

False

>>> 'Hello world!'.startswith('Hello world!')

True

>>> 'Hello world!'.endswith('Hello world!')

True
```

These methods are useful alternatives to the == equals operator if you need to check only whether the first or last part of the string, rather than the whole thing, is equal to another string.

The join() and split() String Methods

The `join()` method is useful when you have a list of strings that need to be joined together into a single string value. The `join()` method is called on a string, gets passed a list of strings, and returns a string. The returned string is the concatenation of each string in the passed-in list. For example, enter the following into the interactive shell:

```
>>> ', '.join(['cats', 'rats', 'bats'])

'cats, rats, bats'

>>> ' '.join(['My', 'name', 'is', 'Simon'])

'My name is Simon'

>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])

'MyABCnameABCisABCSimon'
```

Notice that the string `join()` calls on is inserted between each string of the list argument. For example, when `join(['cats', 'rats', 'bats'])` is called on the `', '` string, the returned string is 'cats, rats, bats'.

Remember that `join()` is called on a string value and is passed a list value. (It's easy to accidentally call it the other way around.) The `split()` method does the opposite: It's called on a string value and returns a list of strings. Enter the following into the interactive shell:

```
>>> 'My name is Simon'.split()

['My', 'name', 'is', 'Simon']
```

By default, the string `'My name is Simon'` is split wherever whitespace characters such as the space, tab, or newline characters are found. These whitespace characters are not included in the strings in the returned list. You can pass a delimiter string to the `split()` method to specify a different string to split upon. For example, enter the following into the interactive shell:

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')

['My', 'name', 'is', 'Simon']

>>> 'My name is Simon'.split('m')

['My na', 'e is Si', 'on']
```

A common use of `split()` is to split a multiline string along the newline characters. Enter the following into the interactive shell:

```
>>> spam = '''Dear  Alice,
How have you been? I am  fine.

There is a container in the fridge
that is labeled "Milk Experiment".


Please do not drink it.
Sincerely,

Bob'''

>>> spam.split('\n')

['Dear Alice,', 'How have you been? I am fine.', 'There is a container in the
fridge', 'that is labeled "Milk Experiment".', '', 'Please do not drink it.',
'Sincerely,', 'Bob']
```

Passing `split()` the argument `'\n'` lets us split the multiline string stored in `spam` along the newlines and return a list in which each item corresponds to one line of the string.

Justifying Text with rjust(), ljust(), and center()

The `rjust()` and `ljust()` string methods return a padded version of the string they are called on, with spaces inserted to justify the text. The first argument to both methods is an integer length for the justified string. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(10)
```

```
'     Hello'
>>> 'Hello'.rjust(20)

'               Hello'
>>> 'Hello World'.rjust(20)

'         Hello World'
>>> 'Hello'.ljust(10)

'Hello'
```

`'Hello'.rjust(10)` says that we want to right-justify `'Hello'` in a string of total length

`10`.`'Hello'` is five characters, so five spaces will be added to its left, giving us a string of 10 characters with `'Hello'` justified right.

An optional second argument to `rjust()` and `ljust()` will specify a fill character other than a space character. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(20, '*')

'***************Hello'

>>> 'Hello'.ljust(20, '-')

'Hello --------------- '
```

The `center()` string method works like `ljust()` and `rjust()` but centers the text rather than justifying it to the left or right. Enter the following into the interactive shell:

```
>>> 'Hello'.center(20)

'       Hello        '

>>> 'Hello'.center(20, '=')

'=======Hello========'
```

These methods are especially useful when you need to print tabular data that has the correct spacing. Open a new file editor window and enter the following code, saving it as *picnicTable.py*:

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():

        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))
picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies':  8000}

printPicnic(picnicItems, 12, 5)

printPicnic(picnicItems, 20, 6)
```

In this program, we define a `printPicnic()` method that will take in a dictionary of information and use `center()`, `ljust()`, and `rjust()` to display that information in a neatly aligned table-like format.

The dictionary that we'll pass to `printPicnic()` is `picnicItems`. In `picnicItems`, we have 4 sandwiches, 12 apples, 4 cups, and 8000 cookies. We want to organize this information into two columns, with the name of the item on the left and the quantity on the right.

To do this, we decide how wide we want the left and right columns to be. Along with our dictionary, we'll pass these values to `printPicnic()`.

`printPicnic()` takes in a dictionary, a `leftWidth` for the left column of a table, and a `rightWidth` for the right column. It prints a title, `PICNIC ITEMS`, centered above the table. Then, it loops through the dictionary, printing each key-value pair on a line with the key justified left and padded by periods, and the value justified right and padded by spaces.

After defining `printPicnic()`, we define the dictionary `picnicItems` and call printPicnic() twice, passing it different widths for the left and right table columns.

When you run this program, the picnic items are displayed twice. The first time the left column is 12 characters wide, and the right column is 5 characters wide. The second time they are 20 and 6 characters wide, respectively.

```
---PICNIC ITEMS--

sandwiches..    4

apples ......12

cups…....    4

cookies…..  8000

-------PICNIC ITEMS-------
sandwiches….......     4
apples…...........    12
cups….............     4
cookies…..........  8000
```

Using rjust(), ljust(), and center() lets you ensure that strings are neatly aligned, even if you aren't sure how many characters long your strings are.

Removing Whitespace with strip(), rstrip(), and lstrip()

Sometimes you may want to strip off whitespace characters (space, tab, and newline) from the left side, right side, or both sides of a string. The strip() string method will return a new string without any whitespace characters at the beginning or end. The lstrip() and rstrip() methods will remove whitespace characters from the left and right ends, respectively. Enter the following into the interactive shell:

```
>>> spam = '    Hello World     '

>>> spam.strip()

'Hello World'

>>> spam.lstrip()

'Hello World '

>>> spam.rstrip()

'    Hello World'
```

Optionally, a string argument will specify which characters on the ends should be stripped. Enter the following into the interactive shell:

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'

>>> spam.strip('ampS')

'BaconSpamEggs'
```

Passing strip() the argument 'ampS' will tell it to strip occurences of a, m, p, and capital S from the ends of the string stored in spam. The order of the characters in the string passed to strip() does not matter: strip('ampS') will do the same thing as strip('mapS') or strip('Spam').

Copying and Pasting Strings with the pyperclip Module

The pyperclip module has copy() and paste() functions that can send text to and receive text from your computer's clipboard. Sending the output of your program to the clipboard will make it easy to paste it to an email, word processor, or some other software.

Pyperclip does not come with Python. To install it, follow the directions for installing third-party modules in Appendix A. After installing the pyperclip module, enter the following into the interactive shell:

```
>>> import pyperclip
>>> pyperclip.copy('Hello world!')

>>> pyperclip.paste()

'Hello world!'
```

Of course, if something outside of your program changes the clipboard contents, the `paste()` function will return it. For example, if I copied this sentence to the clipboard and then called `paste()`, it would look like this:

```
>>> pyperclip.paste()

'For example, if I copied this sentence to the clipboard and then called
paste(), it would look like this:'
```

RUNNING PYTHON SCRIPTS OUTSIDE OF IDLE

So far, you've been running your Python scripts using the interactive shell and file editor in IDLE. However, you won't want to go through the inconvenience of opening IDLE and the Python script each time you want to run a script. Fortunately, there are shortcuts you can set up to make running Python scripts easier. The steps are slightly different for Windows, OS X, and Linux, but each is described in Appendix B. Turn to Appendix B to learn how to run your Python scripts conveniently and be able to pass command line arguments to them. (You will not be able to pass command line arguments to your programs using IDLE.)

Project: Password Locker

You probably have accounts on many different websites. It's a bad habit to use the same password for each of them because if any of those sites has a security breach, the hackers will learn the password to all of your other accounts. It's best to use password manager software on your computer that uses one master password to unlock the password manager. Then you can copy any account password to the clipboard and paste it into the website's Password field.

The password manager program you'll create in this example isn't secure, but it offers a basic demonstration of how such programs work.

THE CHAPTER PROJECTS

This is the first "chapter project" of the book. From here on, each chapter will have projects that demonstrate the concepts covered in the chapter. The projects are written in a style that takes you from a blank file editor window to a full, working program. Just like with the interactive shell examples, don't only read the project sections — follow along on your computer!

Step 1: Program Design and Data Structures

You want to be able to run this program with a command line argument that is the account's name — for instance, *email* or *blog*. That account's password will be copied to the clipboard so that the user can paste it into a Password field. This way, the user can have long, complicated passwords without having to memorize them.

Open a new file editor window and save the program as *pw.py*. You need to start the program with a #! (*shebang*) line (see Appendix B) and should also write a comment that briefly describes the program. Since you want to associate each account's name with its password, you can store these as strings in a dictionary. The dictionary will be the data structure that organizes your account and password data. Make your program look like the following:

```
#! python3

# pw.py - An insecure password locker program.


PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}
```

Step 2: Handle Command Line Arguments

The command line arguments will be stored in the variable `sys.argv`. (See Appendix B for more information on how to use command line arguments in your programs.) The first item in the `sys.argv` list should always be a string containing the program's filename (`'pw.py'`), and the second item should be the first command line argument. For this program, this argument is the name of the account whose password you want. Since the command line argument is mandatory, you display a usage message to the user if they forget to add it (that is, if the `sys.argv` list has fewer than two values in it). Make your program look like the following:

```
#! python3

# pw.py - An insecure password locker program.


PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',
             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}

import sys

if len(sys.argv) < 2:

    print('Usage: python pw.py [account] - copy account password')
    sys.exit()
```

```
account = sys.argv[1]      # first command line arg is the account name
```

Step 3: Copy the Right Password

Now that the account name is stored as a string in the variable `account`, you need to see whether it exists in the `PASSWORDS` dictionary as a key. If so, you want to copy the key's value to the clipboard using `pyperclip.copy()`. (Since you're using the `pyperclip` module, you need to import it.) Note that you don't actually *need* the `account` variable; you could just use `sys.argv[1]` everywhere `account` is used in this program. But a variable named `account` is much more readable than something cryptic like `sys.argv[1]`.

Make your program look like the following:

```
#! python3

# pw.py - An insecure password locker program.
PASSWORDS = {'email': 'F7minlBDDuvMJuxESSKHFhTxFtjVB6',

             'blog': 'VmALvQyKAxiVH5G8v01if1MLZF3sdt',
             'luggage': '12345'}


import sys, pyperclip

if len(sys.argv) < 2:

    print('Usage: py pw.py [account] - copy account password')
    sys.exit()


account = sys.argv[1]   # first command line arg is the account name


if account in PASSWORDS:
    pyperclip.copy(PASSWORDS[account])

    print('Password for ' + account + ' copied to clipboard.')
else:

    print('There is no account named ' + account)
```

This new code looks in the `PASSWORDS` dictionary for the account name. If the account name is a key in the dictionary, we get the value corresponding to that key, copy it to the clipboard, and print a message saying that we copied the value. Otherwise, we print a message saying there's no account with that name.

That's the complete script. Using the instructions in Appendix B for launching command line programs easily, you now have a fast way to copy your account passwords to the clipboard. You will have to modify the `PASSWORDS` dictionary value in the source whenever you want to update the program with a new password.

Of course, you probably don't want to keep all your passwords in one place where anyone could easily copy them. But you can modify this program and use it to quickly copy regular text to the clipboard. Say you are sending out several emails that have many of the same stock paragraphs in common. You could put each paragraph as a value in the `PASSWORDS` dictionary (you'd probably want to rename the dictionary at this point), and then you would have a way to quickly select and copy one of many standard pieces of text to the clipboard.

On Windows, you can create a batch file to run this program with the WIN-R Run window. (For more about batch files, see Appendix B.) Type the following into the file editor and save the file as *pw.bat* in the *C:\Windows* folder:

```
@py.exe C:\Python34\pw.py %*
@pause
```

With this batch file created, running the password-safe program on Windows is just a matter of pressing WIN-R and typing `pw <account name>`.

Project: Adding Bullets to Wiki Markup

When editing a Wikipedia article, you can create a bulleted list by putting each list item on its own line and placing a star in front. But say you have a really large list that you want to add bullet points to. You could just type those stars at the beginning of each line, one by one. Or you could automate this task with a short Python script.

The *bulletPointAdder.py* script will get the text from the clipboard, add a star and space to the beginning of each line, and then paste this new text to the clipboard. For example, if I copied the following text (for the Wikipedia article "List of Lists of Lists") to the clipboard:

```
Lists of  animals
Lists of aquarium life

Lists of biologists by author abbreviation
Lists of cultivars
```

and then ran the *bulletPointAdder.py* program, the clipboard would then contain the following:

```
* Lists of animals
* Lists of aquarium life
* Lists of biologists by author abbreviation
* Lists of cultivars
```

This star-prefixed text is ready to be pasted into a Wikipedia article as a bulleted list.

Step 1: Copy and Paste from the Clipboard

You want the *bulletPointAdder.py* program to do the following:

1. Paste text from the clipboard
2. Do something to it
3. Copy the new text to the clipboard

That second step is a little tricky, but steps 1 and 3 are pretty straightforward: They just involve the `pyperclip.copy()` and `pyperclip.paste()` functions. For now, let's just write the part of the program that covers steps 1 and 3. Enter the following, saving the program as *bulletPointAdder.py*:

```
#! python3

# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.


import pyperclip

text = pyperclip.paste()


# TODO: Separate lines and add stars.

pyperclip.copy(text)
```

The TODOcomment is a reminder that you should complete this part of the program eventually. The next step is to actually implement that piece of the program.

Step 2: Separate the Lines of Text and Add the Star

The call to pyperclip.paste()returns all the text on the clipboard as one big string. If we used the "List of Lists of Lists" example, the string stored in textwould look like this:

```
'Lists of animals\nLists of aquarium life\nLists of biologists by author
abbreviation\nLists of cultivars'
```

The \nnewline characters in this string cause it to be displayed with multiple lines when it is printed or pasted from the clipboard. There are many “lines” in this one string value.
You want to add a star to the start of each of these lines.

You could write code that searches for each \n newline character in the string and then adds the star just after that. But it would be easier to use the split()method to return a list of strings, one for each line in the original string, and then add the star to the front of each string in the list.

Make your program look like the following:

```
#! python3

# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.


import pyperclip

text = pyperclip.paste()


# Separate lines and add stars.
lines = text.split('\n')

for i in range(len(lines)):     # loop through all indexes in the "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines"  list


pyperclip.copy(text)
```

We split the text along its newlines to get a list in which each item is one line of the text. We store the list in `lines` and then loop through the items in `lines`. For each line, we add a star and a space to the start of the line. Now each string in `lines` begins with a star.

Step 3: Join the Modified Lines

The `lines` list now contains modified lines that start with stars. But `pyperclip.copy()` is expecting a single string value, not a list of string values. To make this single string value, pass `lines` into the `join()` method to get a single string joined from the list’s strings.
Make your program look like the following:

```
#! python3

# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.


import pyperclip

text = pyperclip.paste()


# Separate lines and add stars.
lines = text.split('\n')

for i in range(len(lines)):     # loop through all indexes for "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines"  list

text = '\n'.join(lines)

pyperclip.copy(text)
```

When this program is run, it replaces the text on the clipboard with text that has stars at the start of each line. Now the program is complete, and you can try running it with text copied to the clipboard.

Even if you don’t need to automate this specific task, you might want to automate some other kind of text manipulation, such as removing trailing spaces from the end of lines or converting text to uppercase or lowercase. Whatever your needs, you can use the clipboard for input and output.