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 writing programs that handle large amounts of data easier. 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 sequence data types (lists, tuples, and strings) and show how they compare with each other. 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 `spam` variable 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. Note that because the first index is `0`, the last index is one less than the size of the list; a list of four items has `3` as its last index.

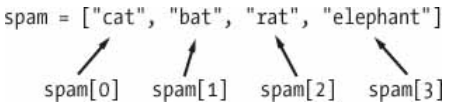

Figure 4-1: A list value stored in the variable `spam`, showing which value each index refers to

For example, enter 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'
❶ >>> 'Hello, ' + spam[0]
❷ 'Hello, cat'


>>> 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
'The bat ate the cat.'
```

Notice that the expression `'Hello, ' + spam[0]` ❶ evaluates to `'Hello, ' + 'cat'` because `spam[0]` evaluates to the string `'cat'`. This expression in turn evaluates to the string value `'Hello, cat'` ❷.

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 or slices, 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 a List from Another List 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 the len() Function

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

Lists can be concatenated and replicated just like strings. The `+` operator combines two lists to create a new list value and the `*` operator can 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'
```

It turns out that this is a bad way to write code. (Also, I don't actually own this many cats, I swear.) 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, enter 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
```

You can view the execution of these programs at *https://autbor.com/allmycats1/* and *https://autbor.com/allmycats2/*. 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 item in a list 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 sequence value that Python considers similar to `[0, 1, 2, 3]`. (Sequences are described in "Sequence Data Types" on page 93.) The following program has the same output as the previous one:

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

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

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', 'flamethrowers', '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: flamethrowers
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
```

You can view the execution of this program at *https://autbor.com/mypets/*.

The Multiple Assignment Trick

The *multiple assignment trick* (technically called *tuple unpacking*) 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', 'gray', 'loud']
>>> size = cat[0]
>>> color = cat[1]
>>> disposition = cat[2]
```

you could type this line of code:

```
>>> cat = ['fat', 'gray', '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', 'gray', 'loud']
>>> size, color, disposition, name = cat
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    size, color, disposition, name = cat
ValueError: not enough values to unpack (expected 4, got 3)
```

Using the enumerate() Function with Lists

Instead of using the `range(len(someList))` technique with a `for` loop to obtain the integer index of the items in the list, you can call the `enumerate()` function instead. On each iteration of the loop, `enumerate()` will return two values: the index of the item in the list, and the item in the list itself. For example, this code is equivalent to the code in the “Using for Loops with Lists” on page 84:

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
>>> for index, item in enumerate(supplies):
...     print('Index ' + str(index) + ' in supplies is: ' + item)

Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flamethrowers
Index 3 in supplies is: binders
```

The `enumerate()` function is useful if you need both the item and the item’s index in the loop’s block.

Using the random.choice() and random.shuffle() Functions with Lists

The `random` module has a couple functions that accept lists for arguments. The `random.choice()` function will return a randomly selected item from the list. Enter the following into the interactive shell:

```
>>> import random
>>> pets = ['Dog', 'Cat', 'Moose']
>>> random.choice(pets)
'Dog'
>>> random.choice(pets)
'Cat'
>>> random.choice(pets)
'Cat'
```

You can consider `random.choice(someList)` to be a shorter form of `someList[random.randint(0, len(someList) - 1]`.

The `random.shuffle()` function will reorder the items in a list. This function modifies the list in place, rather than returning a new list. Enter the following into the interactive shell:

```
>>> import random
>>> people = ['Alice', 'Bob', 'Carol', 'David']
>>> random.shuffle(people)
>>> people
['Carol', 'David', 'Alice', 'Bob']
>>> random.shuffle(people)

>>> people
['Alice', 'David', 'Bob', 'Carol']
```

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 += 1`	`spam = spam + 1`

Augmented assignment statement	Equivalent assignment statement
`spam -= 1`	`spam = spam - 1`
`spam *= 1`	`spam = spam * 1`
`spam /= 1`	`spam = spam / 1`
`spam %= 1`	`spam = 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 shortly) 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 `spam` as 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" on page 94.

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 `AttributeError` error 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 the remove() Method

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 useful 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. Enter 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: '<' not supported between instances of 'str' and '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.

Reversing the Values in a List with the reverse() Method

If you need to quickly reverse the order of the items in a list, you can call the `reverse()` list method. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'moose']
>>> spam.reverse()


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

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 does not matter; Python knows that the list is not finished until it sees the ending square bracket. 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:

```
print('Four score and seven ' + \
      'years ago...')
```

These tricks are useful when you want to rearrange long lines of Python code to be a bit more readable.

Like the `sort()` list method, `reverse()` doesn't return a list. This is why you write `spam.reverse()`, instead of `spam = spam.reverse()`.

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.randint(0, len(messages) - 1)])
```

You can view the execution of this program at *https://autbor.com/magic8ball2/*.

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 for `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 `messages` list 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.

Sequence Data Types

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. The Python sequence data types include lists, strings, range objects returned by `range()`, and tuples (explained in the "The Tuple Data Type" on page 96). Many of the things you can do with lists can also be done with strings and other values of sequence types: 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]
```

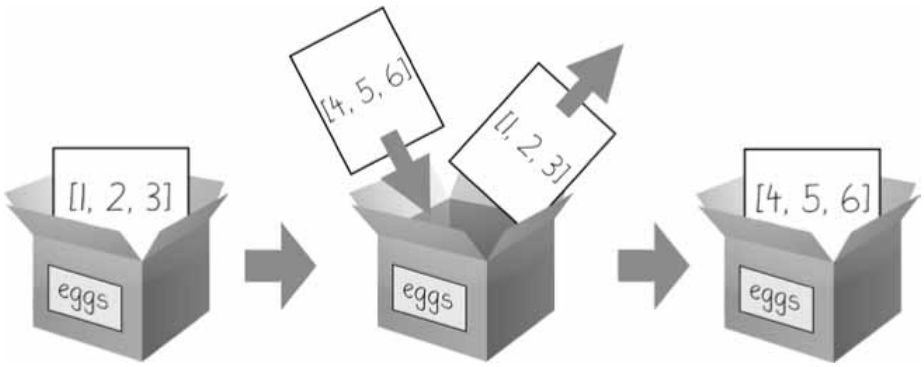

Figure 4-2: When `eggs = [4, 5, 6]` is executed, the contents of `eggs` are replaced with a new list value.

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.

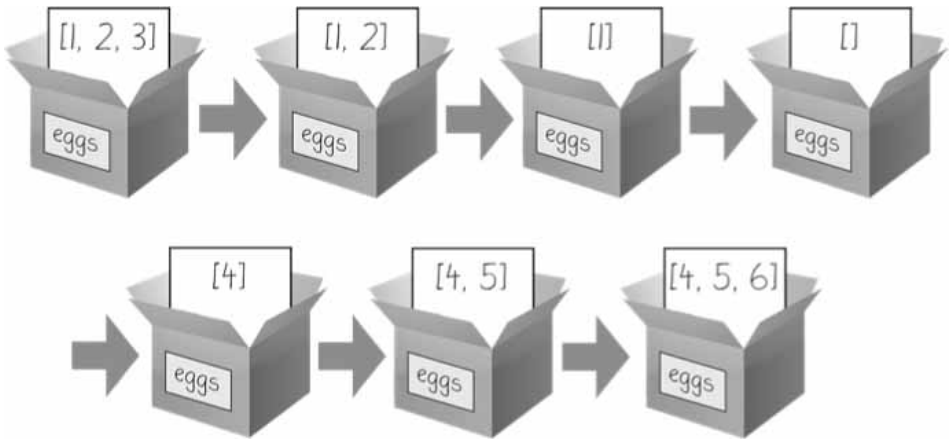

Figure 4-3: The `del` statement and the `append()` method modify the same list value in place.

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" on page 100 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 `TypeError` error 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, it's fine to have a trailing comma after the last item in a list or tuple in Python.) 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. However, this explanation is a simplification of what Python is actually doing. Technically, variables are storing references to the computer memory locations where the values are stored. Enter the following into the interactive shell:

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

When you assign `42` to the `spam` variable, you are actually creating the `42` value in the computer's memory and storing a *reference* to it in the spam variable. When you copy the value in `spam` and assign it to the variable `cheese`, you are actually copying the reference. Both the `spam` and `cheese` variables refer to the `42` value in the computer's memory. When you later change the value in `spam` to `100`, you're creating a new `100` value and storing a reference to it in `spam`. This doesn't affect the value in `cheese`. Integers are *immutable* values that don't change; changing the *spam* variable is actually making it refer to a completely different value in memory.

But lists don't work this way, because list values can change; that is, lists are *mutable*. 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 # The reference is being copied, not the list.


❸ >>> cheese[1] = 'Hello!' # This changes the list value.
  >>> spam
  [0, 'Hello!', 2, 3, 4, 5]
  >>> cheese # The cheese variable refers to the same list.
  [0, 'Hello!', 2, 3, 4, 5]
```

This might look odd to you. The code touched only the `cheese` list, but it seems that both the `cheese` and `spam` lists 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.

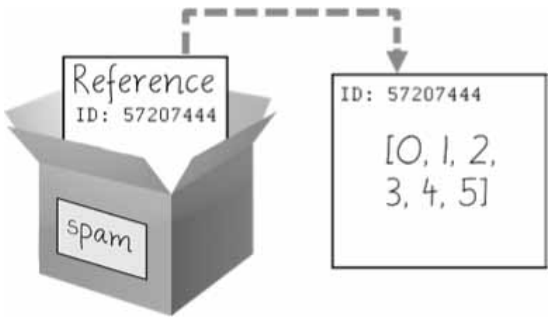

Figure 4-4: `spam = [0, 1, 2, 3, 4, 5]` stores a reference to a list, not the actual list.

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.

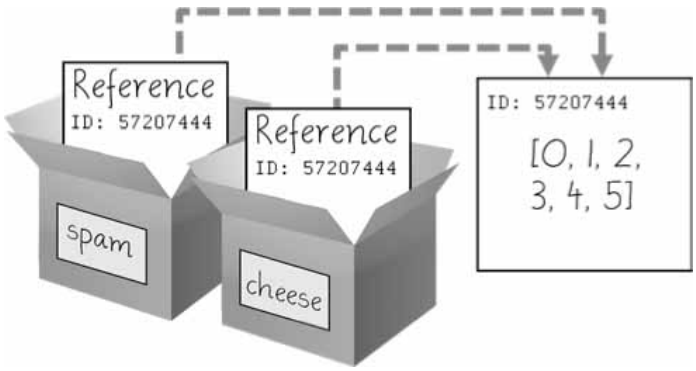

Figure 4-5: `spam = cheese` copies the reference, not the list.

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.

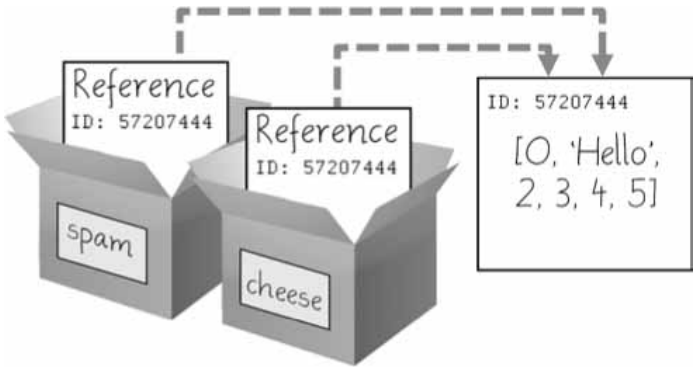

Figure 4-6: `cheese[1] = 'Hello!'` modifies the list that both variables refer to.

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

Identity and the id() Function

You may be wondering why the weird behavior with mutable lists in the previous section doesn't happen with immutable values like integers or strings. We can use Python's `id()` function to understand this. All values in Python have a unique identity that can be obtained with the `id()` function. Enter the following into the interactive shell:

```
>>> id('Howdy') # The returned number will be different on your machine.
44491136
```

When Python runs `id('Howdy')`, it creates the `'Howdy'` string in the computer's memory. The numeric memory address where the string is stored is returned by the `id()` function. Python picks this address based on which memory bytes happen to be free on your computer at the time, so it'll be different each time you run this code.

Like all strings, `'Howdy'` is immutable and cannot be changed. If you "change" the string in a variable, a new string object is being made at a different place in memory, and the variable refers to this new string. For example, enter the following into the interactive shell and see how the identity of the string referred to by `bacon` changes:

```
>>> bacon = 'Hello'
>>> id(bacon)
44491136
>>> bacon += ' world!' # A new string is made from 'Hello' and ' world!'.
>>> id(bacon) # bacon now refers to a completely different string.
44609712
```

However, lists can be modified because they are mutable objects. The `append()` method doesn't create a new list object; it changes the existing list object. We call this "modifying the object *in-place.*"

```
>>> eggs = ['cat', 'dog'] # This creates a new list.
>>> id(eggs)
35152584
>>> eggs.append('moose') # append() modifies the list "in place".
>>> id(eggs) # eggs still refers to the same list as before.
35152584
>>> eggs = ['bat', 'rat', 'cow'] # This creates a new list, which has a new
identity.
>>> id(eggs) # eggs now refers to a completely different list.
44409800
```

If two variables refer to the same list (like `spam` and `cheese` in the previous section) and the list value itself changes, both variables are affected because they both refer to the same list. The `append()`, `extend()`, `remove()`, `sort()`, `reverse()`, and other list methods modify their lists in place.

Python's *automatic garbage collector* deletes any values not being referred to by any variables to free up memory. You don't need to worry about how the garbage collector works, which is a good thing: manual memory management in other programming languages is a common source of bugs.

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']
>>> id(spam)
44684232
>>> cheese = copy.copy(spam)
>>> id(cheese) # cheese is a different list with different identity.
44685832
>>> 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 `1`. 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.

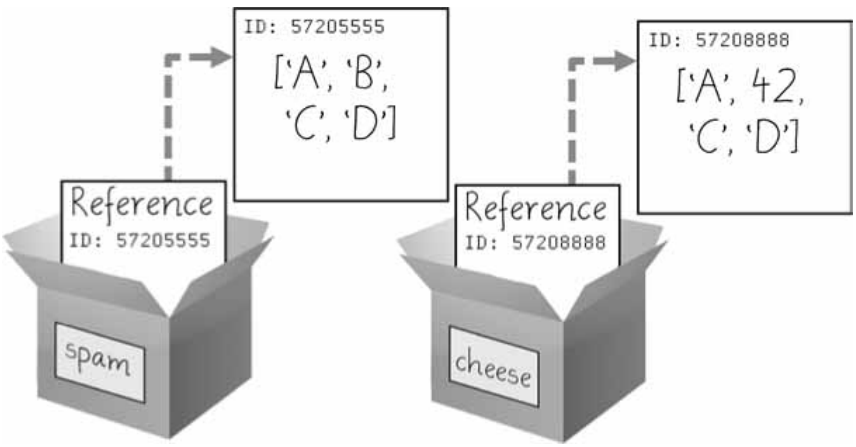

Figure 4-7: `cheese = copy.copy(spam)` creates a second list that can be modified independently of the first.

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.

A Short Program: Conway's Game of Life

Conway's Game of Life is an example of *cellular automata*: a set of rules governing the behavior of a field made up of discrete cells. In practice, it creates a pretty animation to

look at. You can draw out each step on graph paper, using the squares as cells. A filled-in square will be “alive” and an empty square will be “dead.” If a living square has two or three living neighbors, it continues to live on the next step. If a dead square has exactly three living neighbors, it comes alive on the next step. Every other square dies or remains dead on the next step. You can see an example of the progression of steps in Figure 4-8.

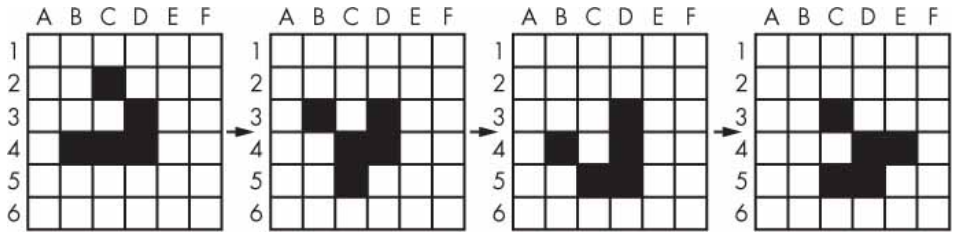

Figure 4-8: Four steps in a Conway’s Game of Life simulation

Even though the rules are simple, there are many surprising behaviors that emerge. Patterns in Conway’s Game of Life can move, self-replicate, or even mimic CPUs. But at the foundation of all of this complex, advanced behavior is a rather simple program.

We can use a list of lists to represent the two-dimensional field. The inner list represents each column of squares and stores a `'#'` hash string for living squares and a `' '` space string for dead squares. Type the following source code into the file editor, and save the file as *conway.py*. It’s fine if you don’t quite understand how all of the code works; just enter it and follow along with comments and explanations provided here as close as you can:

```
# Conway's Game of Life
import random, time, copy
WIDTH = 60

HEIGHT = 20

# Create a list of list for the cells:
nextCells = []
for x in range(WIDTH):
    column = [] # Create a new column.
    for y in range(HEIGHT):
        if random.randint(0, 1) == 0:
            column.append('#') # Add a living cell.
        else:
            column.append(' ') # Add a dead cell.
    nextCells.append(column) # nextCells is a list of column lists.

while True: # Main program loop.
    print('\n\n\n\n\n') # Separate each step with newlines.
    currentCells = copy.deepcopy(nextCells)

    # Print currentCells on the screen:
    for y in range(HEIGHT):
        for x in range(WIDTH):
            print(currentCells[x][y], end='') # Print the # or space.
        print() # Print a newline at the end of the row.

    # Calculate the next step's cells based on current step's cells:
    for x in range(WIDTH):
        for y in range(HEIGHT):
            # Get neighboring coordinates:
```

```
            # `% WIDTH` ensures leftCoord is always between 0 and WIDTH - 1
            leftCoord  = (x - 1) % WIDTH
            rightCoord = (x + 1) % WIDTH
            aboveCoord = (y - 1) % HEIGHT
            belowCoord = (y + 1) % HEIGHT

            # Count number of living neighbors:
            numNeighbors = 0
            if currentCells[leftCoord][aboveCoord] == '#':
                numNeighbors += 1 # Top-left neighbor is alive.
            if currentCells[x][aboveCoord] == '#':
                numNeighbors += 1 # Top neighbor is alive.
            if currentCells[rightCoord][aboveCoord] == '#':
                numNeighbors += 1 # Top-right neighbor is alive.
            if currentCells[leftCoord][y] == '#':
                numNeighbors += 1 # Left neighbor is alive.
            if currentCells[rightCoord][y] == '#':
                numNeighbors += 1 # Right neighbor is alive.
            if currentCells[leftCoord][belowCoord] == '#':
                numNeighbors += 1 # Bottom-left neighbor is alive.
            if currentCells[x][belowCoord] == '#':
                numNeighbors += 1 # Bottom neighbor is alive.
            if currentCells[rightCoord][belowCoord] == '#':
                numNeighbors += 1 # Bottom-right neighbor is alive.

            # Set cell based on Conway's Game of Life rules:
            if currentCells[x][y] == '#' and (numNeighbors == 2 or
numNeighbors == 3):
                # Living cells with 2 or 3 neighbors stay alive:
                nextCells[x][y] = '#'
            elif currentCells[x][y] == ' ' and numNeighbors == 3:
                # Dead cells with 3 neighbors become alive:
                nextCells[x][y] = '#'
            else:
                # Everything else dies or stays dead:
                nextCells[x][y] = ' '
    time.sleep(1) # Add a 1-second pause to reduce flickering.
```

Let's look at this code line by line, starting at the top.

```
# Conway's Game of Life
import random, time, copy
WIDTH = 60
HEIGHT = 20
```

First we import modules that contain functions we'll need, namely the `random.randint()`, `time.sleep()`, and `copy.deepcopy()` functions.

```
# Create a list of list for the cells:
nextCells = []
for x in range(WIDTH):
    column = [] # Create a new column.
    for y in range(HEIGHT):
        if random.randint(0, 1) == 0:
            column.append('#') # Add a living cell.
```

```
        else:
            column.append(' ') # Add a dead cell.
    nextCells.append(column) # nextCells is a list of column lists.
```

The very first step of our cellular automata will be completely random. We need to create a list of lists data structure to store the `'#'` and `' '` strings that represent a living or dead cell, and their place in the list of lists reflects their position on the screen. The inner lists each represent a column of cells. The `random.randint(0, 1)` call gives an even 50/50 chance between the cell starting off alive or dead.

We put this list of lists in a variable called `nextCells`, because the first step in our main program loop will be to copy `nextCells` into `currentCells`. For our list of lists data structure, the x-coordinates start at 0 on the left and increase going right, while the y-coordinates start at 0 at the top and increase going down. So `nextCells[0][0]` will represent the cell at the top left of the screen, while `nextCells[1][0]` represents the cell to the right of that cell and `nextCells[0][1]` represents the cell beneath it.

```
while True: # Main program loop.
    print('\n\n\n\n\n') # Separate each step with newlines.
    currentCells = copy.deepcopy(nextCells)
```

Each iteration of our main program loop will be a single step of our cellular automata. On each step, we'll copy `nextCells` to `currentCells`, print `currentCells` on the screen, and then use the cells in `currentCells` to calculate the cells in `nextCells`.

```
    # Print currentCells on the screen:
    for y in range(HEIGHT):
        for x in range(WIDTH):


            print(currentCells[x][y], end='') # Print the # or space.
        print() # Print a newline at the end of the row.
```

These nested `for` loops ensure that we print a full row of cells to the screen, followed by a newline character at the end of the row. We repeat this for each row in `nextCells`.

```
    # Calculate the next step's cells based on current step's cells:
    for x in range(WIDTH):
        for y in range(HEIGHT):
            # Get neighboring coordinates:
            # `% WIDTH` ensures leftCoord is always between 0 and WIDTH - 1
            leftCoord  = (x - 1) % WIDTH
            rightCoord = (x + 1) % WIDTH
            aboveCoord = (y - 1) % HEIGHT
            belowCoord = (y + 1) % HEIGHT
```

Next, we need to use two nested `for` loops to calculate each cell for the next step. The living or dead state of the cell depends on the neighbors, so let's first calculate the index of the cells to the left, right, above, and below the current x- and y-coordinates.

The `%` mod operator performs a "wraparound." The left neighbor of a cell in the leftmost column `0` would be `0 - 1` or `-1`. To wrap this around to the rightmost column's index, `59`, we calculate `(0 - 1) % WIDTH`. Since `WIDTH` is `60`, this expression evaluates to `59`. This mod-wraparound technique works for the right, above, and below neighbors as well.

```
            # Count number of living neighbors:
            numNeighbors = 0
            if currentCells[leftCoord][aboveCoord] == '#':
```

```
                numNeighbors += 1 # Top-left neighbor is alive.
            if currentCells[x][aboveCoord] == '#':
                numNeighbors += 1 # Top neighbor is alive.
            if currentCells[rightCoord][aboveCoord] == '#':
                numNeighbors += 1 # Top-right neighbor is alive.
            if currentCells[leftCoord][y] == '#':
                numNeighbors += 1 # Left neighbor is alive.
            if currentCells[rightCoord][y] == '#':
                numNeighbors += 1 # Right neighbor is alive.
            if currentCells[leftCoord][belowCoord] == '#':
                numNeighbors += 1 # Bottom-left neighbor is alive.
            if currentCells[x][belowCoord] == '#':
                numNeighbors += 1 # Bottom neighbor is alive.
            if currentCells[rightCoord][belowCoord] == '#':
                numNeighbors += 1 # Bottom-right neighbor is alive.
```

To decide if the cell at `nextCells[x][y]` should be living or dead, we need to count the number of living neighbors `currentCells[x][y]` has. This series of `if` statements checks each of the eight neighbors of this cell, and adds `1` to `numNeighbors` for each living one.

```
            # Set cell based on Conway's Game of Life rules:
            if currentCells[x][y] == '#' and (numNeighbors == 2 or
numNeighbors == 3):
                # Living cells with 2 or 3 neighbors stay alive:
                nextCells[x][y] = '#'
            elif currentCells[x][y] == ' ' and numNeighbors == 3:
                # Dead cells with 3 neighbors become alive:


                nextCells[x][y] = '#'
            else:
                # Everything else dies or stays dead:
                nextCells[x][y] = ' '
    time.sleep(1) # Add a 1-second pause to reduce flickering.
```

Now that we know the number of living neighbors for the cell at `currentCells[x][y]`, we can set `nextCells[x][y]` to either `'#'` or `' '`. After we loop over every possible x- and y-coordinate, the program takes a 1-second pause by calling `time.sleep(1)`. Then the program execution goes back to the start of the main program loop to continue with the next step.

Several patterns have been discovered with names such as “glider,” “propeller,” or “heavyweight spaceship.” The glider pattern, pictured in Figure 4-8, results in a pattern that “moves” diagonally every four steps. You can create a single glider by replacing this line in our *conway.py* program:

```
        if random.randint(0, 1) == 0:
```

with this line:

```
        if (x, y) in ((1, 0), (2, 1), (0, 2), (1, 2), (2, 2)):
```

You can find out more about the intriguing devices made using Conway’s Game of Life by searching the web. And you can find other short, text-based Python programs like this one at *https://github.com/asweigart/pythonstdiogames*.

SUMMARY

Lists are useful data types since they allow you to write code that works on a modifiable number of values in a single variable. Later in this book, you will see programs using lists to do things that would be difficult or impossible to do without them.

Lists are a sequence data type that is mutable, meaning that their contents can change. Tuples and strings, though also sequence data types, are immutable and cannot be changed. A variable that contains a tuple or string value can be overwritten with a new tuple or string value, but this is not the same thing as modifying the existing value in place—like, say, the `append()` or `remove()` methods do on lists.

Variables do not store list values directly; they store *references* to lists. This is an important distinction when you are copying variables or passing lists as arguments in function calls. Because the value that is being copied is the list reference, be aware that any changes you make to the list might impact another variable in your program. You can use `copy()` or `deepcopy()` if you want to make changes to a list in one variable without modifying the original list.

Practice Questions

1. What is `[]`?
2. How would you assign the value `'hello'` as the third value in a list stored in a variable named `spam`? (Assume `spam` contains `[2, 4, 6, 8, 10]`.)

For the following three questions, let's say `spam` contains the list `['a', 'b', 'c', 'd']`.

3. What does `spam[int(int('3' * 2) // 11)]` evaluate to?
4. What does `spam[-1]` evaluate to?
5. What does `spam[:2]` evaluate to?

For the following three questions, let's say `bacon` contains the list `[3.14, 'cat', 11, 'cat', True]`.

6. What does `bacon.index('cat')` evaluate to?
7. What does `bacon.append(99)` make the list value in `bacon` look like?
8. What does `bacon.remove('cat')` make the list value in `bacon` look like?
9. What are the operators for list concatenation and list replication?
10. What is the difference between the `append()` and `insert()` list methods?
11. What are two ways to remove values from a list?
12. Name a few ways that list values are similar to string values.
13. What is the difference between lists and tuples?
14. How do you type the tuple value that has just the integer value `42` in it?
15. How can you get the tuple form of a list value? How can you get the list form of a tuple value?
16. Variables that "contain" list values don't actually contain lists directly. What do they contain instead?
17. What is the difference between `copy.copy()` and `copy.deepcopy()`?

Practice Projects

For practice, write programs to do the following tasks.

Comma Code

Say you have a list value like this:

```
spam = ['apples', 'bananas', 'tofu', 'cats']
```

Write a function that takes a list value as an argument and returns a string with all the items separated by a comma and a space, with *and* inserted before the last item. For example, passing the previous `spam` list to the function would return `'apples, bananas, tofu, and cats'`. But your function should be able to work with any list value passed to it. Be sure to test the case where an empty list `[]` is passed to your function.

Coin Flip Streaks

For this exercise, we'll try doing an experiment. If you flip a coin 100 times and write down an "H" for each heads and "T" for each tails, you'll create a list that looks like "T T T T H H H H T T." If you ask a human to make up 100 random coin flips, you'll probably end up with alternating head-tail results like "H T H T H H T H T T," which looks random (to humans), but isn't mathematically random. A human will almost never write down a streak of six heads or six tails in a row, even though it is highly likely to happen in truly random coin flips. Humans are predictably bad at being random.

Write a program to find out how often a streak of six heads or a streak of six tails comes up in a randomly generated list of heads and tails. Your program breaks up the experiment into two parts: the first part generates a list of randomly selected `'heads'` and `'tails'` values, and the second part checks if there is a streak in it. Put all of this code in a loop that repeats the experiment 10,000 times so we can find out what percentage of the coin flips contains a streak of six heads or tails in a row. As a hint, the function call `random.randint(0, 1)` will return a `0` value 50% of the time and a `1` value the other 50% of the time.

You can start with the following template:

```
import random

numberOfStreaks = 0

for experimentNumber in range(10000):

    # Code that creates a list of 100 'heads' or 'tails' values.


    # Code that checks if there is a streak of 6 heads or tails in a row.

print('Chance of streak: %s%%' % (numberOfStreaks / 100))
```

Of course, this is only an estimate, but 10,000 is a decent sample size. Some knowledge of mathematics could give you the exact answer and save you the trouble of writing a program, but programmers are notoriously bad at math.

Character Picture Grid

Say you have a list of lists where each value in the inner lists is a one-character string, like this:

```
grid = [['.', '.', '.', '.', '.', '.'],
        ['.', 'O', 'O', '.', '.', '.'],
        ['O', 'O', 'O', 'O', '.', '.'],
        ['O', 'O', 'O', 'O', 'O', '.'],
        ['.', 'O', 'O', 'O', 'O', 'O'],
        ['O', 'O', 'O', 'O', 'O', '.'],
        ['O', 'O', 'O', 'O', '.', '.'],
        ['.', 'O', 'O', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.']]
```

Think of `grid[x][y]` as being the character at the x- and y-coordinates of a "picture" drawn with text characters. The `(0, 0)` origin is in the upper-left corner, the x-coordinates increase going right, and the y-coordinates increase going down.

Copy the previous grid value, and write code that uses it to print the image.

```
..OO.OO..
.OOOOOOO.
.OOOOOOO.
..OOOOO..
...OOO...
....O....
```

Hint: You will need to use a loop in a loop in order to print `grid[0][0]`, then `grid[1][0]`, then `grid[2][0]`, and so on, up to `grid[8][0]`. This will finish the first row, so then print a newline. Then your program should print `grid[0][1]`, then `grid[1][1]`, then `grid[2][1]`, and so on. The last thing your program will print is `grid[8][5]`.

Also, remember to pass the `end` keyword argument to `print()` if you don't want a newline printed automatically after each `print()` call.