6

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 their differences. 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 you can store in a variable or pass to a function, just 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 use quotation marks to mark where the string begins and ends, a list begins with an opening square bracket and ends with a closing square bracket, `[]`.

We call values inside the list *items*. Items are separated with commas (that is, they are *comma-delimited*). For example, enter the following into the interactive shell:

```
>>> [1, 2, 3]  # A list of three integers
[1, 2, 3]
```

```
>>> ['cat', 'bat', 'rat', 'elephant']  # A
list of four strings
['cat', 'bat', 'rat', 'elephant']
>>> ['hello', 3.1415, True, None, 42]  # A
list of several values
['hello', 3.1415, True, None, 42]
❶ >>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam
['cat', 'bat', 'rat', 'elephant']
```

The `spam` variable ❶ is assigned only one value: the list value. But the list value itself contains other values.

Note that the value `[]` is an empty list that contains no values, similar to `''`, the empty string.

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'`, the code `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 6-1 shows a list value assigned to `spam`, along with the index expressions they'd evaluate to. Note that because the first index is `0`, the last index is the size of the list minus one. So, a list of four items has `3` as its last index.

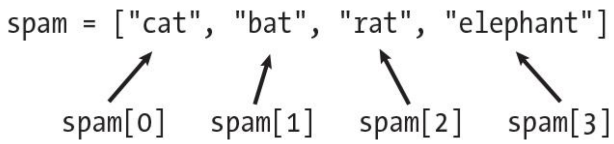

Figure 6-1: A list value stored in the variable spam, showing which value each index refers to Description

For an example of working with indexes, enter the following expressions into the interactive shell. We 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 "<python-input-0>", line 1, in
<module>
    spam[10000]
IndexError: list index out of range
```

Lists can also contain other list values. You can access the values in these lists of lists 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.

Negative Indexes

While indexes start at `0` and go up, you can also use negative integers for the index. For example, enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]  # Last index
'elephant'
>>> spam[-3]  # Third to last index
'bat'
>>> 'The ' + spam[-1] + ' is afraid of the '
+ spam[-3] + '.'
'The elephant is afraid of the bat.'
```

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.

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. We enter a slice between square brackets, like an index, but include 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. The list created from a slice will go up to, but will not include, the value at the second index. For example, 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:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
>>> spam[:]
['cat', 'bat', 'rat', 'elephant']
```

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.

The len() Function

The `len()` function will return the number of values in a list value passed to it. For example, enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'moose']
>>> len(spam)
3
```

This behavior is similar to how the function counts the number of characters in a string value.

Value Updates

Normally, a variable name goes on the left side of an assignment statement, as in spam = 42. However, you can also use an index of a list to change the value at that index:

```
>>> 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]
```

In this example, `spam[1] = 'aardvark'` means "Assign the value at index `1` in the list `spam` to the string `'aardvark'`." You can also use negative indexes like `-1` to update lists.

Concatenation and Replication

You can concatenate and replicate lists with the + and `*` operators, just like strings:

```
>>> [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']
```

The + operator combines two lists to create a new list value, and the `*` operator combines a list and an integer value to replicate the list.

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 operate 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’ll get a `NameError` error because the variable no longer exists. In practice, you almost never need to delete simple variables, however, and the `del` statement is most useful for deleting values from lists.

Working with Lists

When you first begin writing programs, you may be tempted 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 think to write code like this:

```
cat_name_1 = 'Zophie'
cat_name_2 = 'Pooka'
cat_name_3 = 'Simon'
cat_name_4 = 'Lady Macbeth'
```

It turns out that this is a bad way to write code. For one thing, if the number of cats changes (and you can always have more cats), your program will never be able to store more cats than you have variables. These programs also contain a lot of duplicate or nearly identical code. To see this in practice, enter the following program into the file editor and save it as *allMyCats1.py*:

```
print('Enter the name of cat 1:')
cat_name_1 = input()
print('Enter the name of cat 2:')
cat_name_2 = input()
print('Enter the name of cat 3:')
cat_name_3 = input()
print('Enter the name of cat 4:')
cat_name_4 = input()
print('The cat names are:')
print(cat_name_1 + ' ' + cat_name_2 + ' ' +
cat_name_3 + ' ' + cat_name_4)
```

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 enters. In a new file editor window, enter the following source code and save it as *allMyCats2.py*:

```
cat_names = []
while True:
    print('Enter the name of cat ' +
str(len(cat_names) + 1) +
      ' (Or enter nothing to stop.):')
    name = input()
    if name == '':
        break
    cat_names = cat_names + [name]  # List
concatenation
print('The cat names are:')
for name in cat_names:
    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.):

The cat names are:
  Zophie
  Pooka
  Simon
  Lady Macbeth
```

The benefit of using a list is that your data is now in a structure, so your program can process the data much more flexibly than it could with several repetitive variables.

for Loops and Lists

In Chapter 3, 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 to be 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)
```

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(`*some_list*`))` 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 the list 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` occur 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. To see how they work, 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
```

The following program lets the user enter a pet name and then checks 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*:

```
my_pets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()
if name not in my_pets:
    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
```

Keep in mind that the `not in` operator is distinct from the Boolean `not` operator.

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 enter 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 "<python-input-0>", line 1, in
<module>
    size, color, disposition, name = cat
ValueError: not enough values to unpack
(expected 4, got 3)
```

This trick makes your code shorter and more readable than entering three separate lines of code.

List Item Enumeration

Instead of using the `range(len(some_list))` technique with a `for` loop to obtain the integer index of the items in the list, you can call the `enumerate()` function. 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 “for Loops and Lists” on page 115:

```
>>> 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.

Random Selection and Ordering

The `random` module has a couple of 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)
'Cat'
>>> random.choice(pets)
'Cat'
```

```
>>> random.choice(pets)
'Dog'
```

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

The `random.shuffle()` function will reorder the items in a list in place. 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']
```

This function modifies the list in place, rather than returning a new list.

Augmented Assignment Operators

The + and `*` operators that work with strings also work with lists, so let's take a short detour to learn about augmented assignment operators. When assigning a value to a variable, you'll 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 += (which is the regular operator followed by one equal sign) to do the same thing:

```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

There are augmented assignment operators for the +, -, *, /, and % operators, described in Table 6-1.

Table 6-1: The Augmented Assignment Operators

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
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!'  # Same as spam = spam
+ 'world!'
>>> spam
'Hello, world!'
>>> bacon = ['Zophie']
>>> bacon *= 3  # Same as bacon = bacon * 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

Like the multiple assignment trick, augmented assignment operators are a shortcut to make your code simpler and more readable.

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. Think of a method as a function that is always associated with a value. In our `spam` list example, the function would hypothetically be `index(spam, 'hello')`. But since `index()` is a list method and not a function, we call `spam.index('hello')`. Calling `index()` on a list value is how Python knows `index()` is a list method. Let's learn about the list methods in Python.

Finding Values

List values have an `index()` method that can be passed a value. If that value exists in the list, the method will return the index of the value. 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 "<python-input-0>", line 1, in
<module>
    spam.index('howdy howdy howdy')
ValueError: 'howdy howdy howdy' is not in
list
```

When the list contains duplicates of the value, the method returns the index of its first appearance:

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail',
'Pooka']
>>> spam.index('Pooka')
1
```

Notice that `index()` returns `1`, not `3`.

Adding Values

To add new values to a list, use the `append()` and `insert()` methods. The `append()` method adds the argument to the end of the list:

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

The `insert()` method can insert a value at any index in the list. The first argument to `insert()` is the index of 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 doesn't perform any assignment operation, such as `spam = spam.append('moose')` or `spam = spam.insert(1, 'chicken')`. The return value of `append()` and `insert()` is `None`, so you definitely wouldn't want to store it as the new variable value. Rather, these methods modify the list in place, a topic covered in more detail in "Mutable and Immutable Data Types" on page 126.

Methods belong to a single data type. The `append()` and `insert()` methods are list methods, and we can call them on list values only, not on values of other data types, such as strings or integers. To see what happens when we try to do so, enter the following into the interactive shell:

```
>>> eggs = 'hello'
>>> eggs.append('world')
Traceback (most recent call last):
  File "<python-input-0>", 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 "<python-input-0>", line 1, in
<module>
    bacon.insert(1, 'world')
AttributeError: 'int' object has no attribute
'insert'
```

Note the `AttributeError` error messages that show up.

Removing Values

The `remove()` method accepts a value to remove from the list on which it's called:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('bat')
>>> spam
['cat', 'rat', 'elephant']
```

Attempting to delete a value that doesn't exist in the list will result in a `ValueError` error. For example, enter the following into the interactive shell and notice the error it displays:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('chicken')
Traceback (most recent call last):
  File "<python-input-0>", line 1, in
<module>
    spam.remove('chicken')
ValueError: list.remove(x): x not in list
```

If the value appears multiple times in the list, the method will remove only the first instance of it:

```
>>> spam = ['cat', 'bat', 'rat', 'cat',
'hat', 'cat']
>>> spam.remove('cat')
>>> spam
['bat', 'rat', 'cat', 'hat', 'cat']
```

The `del` statement is useful when you know the index of the value you want to remove from the list, while the `remove()` method is useful when you know the value itself.

Sorting Values

You can sort lists of number values or lists of strings 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']
```

The method returns the numbers in numerical order and the strings in alphabetical order. You can also pass `True` as the `reverse` keyword argument to sort the values in reverse order:

```
>>> spam.sort(reverse=True)
>>> spam
['Elephants', 'Dogs', 'Cats', 'Badgers',
'Ants']
```

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

Second, you can't sort lists that have both number values and string values in them, as 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 "<python-input-0>", 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, placing the lowercase *a* 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 argument 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 Values

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']
```

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

EXCEPTIONS TO INDENTATION RULES IN PYTHON

In most cases, the amount of indentation for a line of code tells Python what block it's 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 doesn't matter; Python knows that the list isn't finished until it sees the ending square bracket. This means you can write code that looks like this:

```
spam = ['apples',
 'oranges',
                    'bananas',
'cats']
print(spam[0]) # Prints apples
```

Of course, practically speaking, most people use Python's behavior to make their lists look pretty and readable, like the `messages` list in "A Short Program: Magic 8 Ball with a List" on page 125.

You can also split up a single instruction across multiple lines by ending each line with the *line continuation character* (`\`). Think of `\` as saying, "This instruction continues on the next line." The indentation on the line after a `\` line continuation isn't 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.

Short-Circuiting Boolean Operators

Boolean operators have a subtle behavior that is easy to miss. Recall that if either of the values combined by an `and` operator is `False`, the entire expression is `False`, and if either value combined by an `or` operator is `True`, the entire expression is `True`. If I presented you with the expression `False and spam`, it doesn't matter whether the `spam` variable is `True` or `False` because the entire expression would be `False` either way. The same goes for `True or spam`; this evaluates to `True` no matter the value of `spam`.

Python (and many other programming languages) use this fact to optimize the code so that it runs a little faster by not examining the right-hand side of the Boolean operator at all. This shortcut is called *short-circuiting*. Most of the time, your program will behave the same way it would have if Python checked the entire expression (albeit a few microseconds faster). However, consider this short program, where we check whether the first item in a list is `'cat'`:

```
spam = ['cat', 'dog']
if spam[0] == 'cat':
    print('A cat is the first item.')
else:
    print('The first item is not a cat.')
```

As written, this program prints `A cat is the first item.` But if the list in `spam` is empty, the `spam[0]` code will cause an `IndexError: list Index out of range` error. To fix this, we'll adjust the `if` statement's condition to take advantage of short-circuiting:

```
spam = []
if len(spam) > 0 and spam[0] == 'cat':
    print('A cat is the first item.')
else:
    print('The first item is not a cat.')
```

This program never has an error, because if `len(spam) > 0` is `False` (that is, the list in `spam` is empty), then short-circuiting the `and` operator means that Python doesn't bother running the `spam[0] == 'cat'` code that would cause the `IndexError` error. Keep this short-circuiting behavior in mind when you write code that involves the `and` and `or` operators.

A Short Program: Magic 8 Ball with a List

Using lists, you can write a much more elegant version of Chapter 4's *magic8Ball.py* 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('Ask a yes or no question:')
input('>')
print(messages[random.randint(0,
len(messages) - 1)])
```

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

The `random.randint(0, len(messages) - 1)` call 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 and from the `messages` list without changing other lines of code. If you later update your code, you'll have to change fewer lines, producing fewer chances for you to introduce bugs.

Selecting a random item from a list is common enough that Python has the `random.choice(messages)` function that does the same thing as `random.randint(0, len(messages) - 1)`.

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 Tuple Data Type" on page 127). Many of the things you can do with lists can also be done with strings and other values of sequence types. 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 * * *
```

You can do all the same things with sequence values that you can do with lists: indexing, slicing, `for` loops, `len()`, and the `in` and `not in` operators.

Mutable and Immutable Data Types

But lists and strings differ in an important way. A list value is a *mutable* data type: you can add, remove, or change its values. 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 "<python-input-0>", 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:

```
>>> name = 'Zophie a cat'
>>> new_name = name[0:7] + 'the' + name[8:12]
>>> name
'Zophie a cat'
>>> new_name
'Zophie the cat'
```

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

Although a list value *is* mutable, the second line in the following code doesn't modify the list `eggs`:

```
>>> eggs = ['A', 'B', 'C']
>>> eggs = ['x', 'y', 'z']
```

```
>>> eggs
['x', 'y', 'z']
```

The list value in eggs isn't being changed here; rather, a new and entirely different list value (['x', 'y', 'z']) is replacing the old list value (['A', 'B', 'C']).

If you wanted to actually modify the original list in eggs to contain ['x', 'y', 'z'], you would have to use del statements and the append() method, like this:

```
>>> eggs = ['A', 'B', 'C']
>>> del eggs[2]
>>> del eggs[1]
>>> del eggs[0]
>>> eggs.append('x')
>>> eggs.append('y')
>>> eggs.append('z')
>>> eggs
['x', 'y', 'z']
```

In this example, the eggs variable ends up with the same list value it started with. It's just that this list has been changed (mutated) rather than overwritten. We call this *changing the list in place*.

Mutable versus immutable types may seem like a meaningless distinction, but "References" on page 129 will explain the different behavior when calling functions with mutable arguments versus immutable arguments. First, however, let's find out about the tuple data type, which is an immutable form of the list data type.

The Tuple Data Type

There are only two differences between the *tuple* data type and the list data type. The first difference is that you write tuples using parentheses instead of square brackets. 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
```

The second and primary way that tuples are different from lists is that tuples, like strings, are immutable: you can't modify, append, or remove their values. Enter the following into the interactive shell, and look at the resulting `TypeError` error message:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
Traceback (most recent call last):
  File "<python-input-0>", 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 entered a value inside regular parentheses. (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're immutable and their contents don't change, Python can implement optimizations that make code using tuples slightly faster than code using lists.

List and Tuple Type Conversion

Just as `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

A common metaphor is that variables are boxes that "store" values like strings and integers. However, this explanation is a simplification of what Python is actually doing. A better metaphor is that variables are paper name tags attached to values with string. Enter the following into the interactive shell:

```
❶ >>> spam = 42
❷ >>> eggs = spam
❸ >>> spam = 99
>>> spam
99
>>> eggs
42
```

When you assign 42 to the spam variable, you're 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 eggs, you're copying the reference. Both the spam and eggs variables refer to the 42 value in the computer's memory. Using the name tag metaphor for variables, you've attached the spam name tag and the eggs name tag to the same 42 value. When you assign spam a new 99 value, you've changed what the spam name tag references. Figure 6-2 is a graphical depiction of the code.

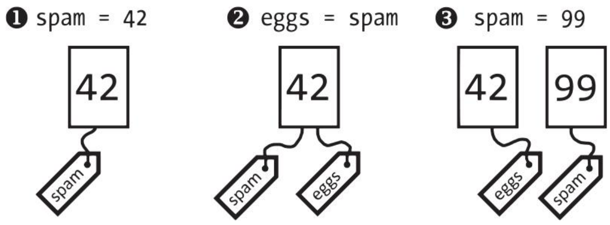

Figure 6-2: Variable assignment doesn't rewrite the value; it changes the reference. Description

The change doesn't affect eggs, which still refers to the 42 value.

But lists don't work this way, because list values can change; that is, lists are *mutable*. Here is code that will make this distinction easier to understand. Enter it into the interactive shell:

```
❶ >>> spam = [0, 1, 2, 3]
❷ >>> eggs = spam  # The reference, not the
  list, is being copied.
❸ >>> eggs[1] = 'Hello!'  # This changes the
  list value.
  >>> spam
  [0, 'Hello!', 2, 3]
  >>> eggs  # The eggs variable refers to the
  same list.
  [0, 'Hello!', 2, 3]
```

This code might look odd to you. It touched only the eggs list, but both the eggs and spam lists seem to have changed.

When you create the list ❶, you assign a reference to it in the spam variable. But the next line copies only the list reference in spam to eggs ❷, not the list value itself. There is still only one list, and spam and eggs now both refer to it. The reason there is only one underlying list is that the list itself was never actually copied. So, when you modify the first element of `eggs` ❸, you're modifying the same list that `spam` refers to. You can see this in Figure 6-3.

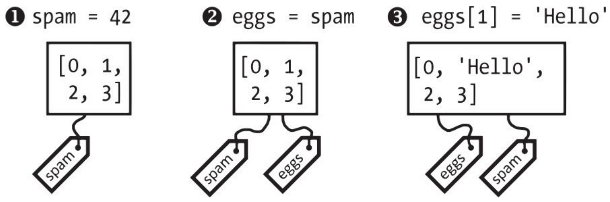

Figure 6-3: Because `spam` and eggs refer to the same list, changing one changes the other. Description

It becomes a bit more complicated, as lists also don't contain a sequence of values directly, but rather a sequence of references to values. I explain this further in "The `copy()` and `deepcopy()` Functions" on page 131.

Although Python variables technically contain references to values, people often casually say that the variable *contains* the value. But keep these two rules in mind:

- In Python, variables never contain values. They contain only references to values.
- In Python, the = assignment operator copies only references. It never copies values.

For the most part, you don't need to know these details, but at times, these simple rules have surprising effects, and you should understand exactly what Python is doing.

Arguments

References are particularly important for understanding how arguments get passed to functions. When a function is called, Python copies to the parameter variables the reference to the arguments. For mutable values like lists (and dictionaries, which I'll describe in Chapter 7), this means the code in the function modifies the original value in place. To see the consequences of this fact, open a new file editor window, enter the following code, and save it as *passingReference.py*:

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


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

```
eggs(spam)
print(spam)  # Prints [1, 2, 3, 'Hello']
```

Notice that when you call `eggs()`, a return value doesn't assign a new value to `spam`. Instead, it directly modifies the list in place. When run, this program outputs `[1, 2, 3, 'Hello']`.

Even though `spam` and `some_parameter` 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 in this way can lead to unexpected behavior and confusing bugs.

The 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 passed to it, you may not want these changes in the original list or dictionary value. To control this behavior, Python provides a module named `copy` that provides both the `copy()` and `deepcopy()` functions. The first of these, `copy.copy()`, can 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']
>>> cheese = copy.copy(spam)  # Creates a
duplicate copy of the list
>>> cheese[1] = 42  # Changes cheese
>>> spam  # The spam variable is unchanged.
['A', 'B', 'C']
>>> cheese  # The cheese variable is changed.
['A', 42, 'C']
```

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`.

Just as variables *refer* to values rather than contain values, lists contain *references* to values rather than values themselves. You can see this in Figure 6-4.

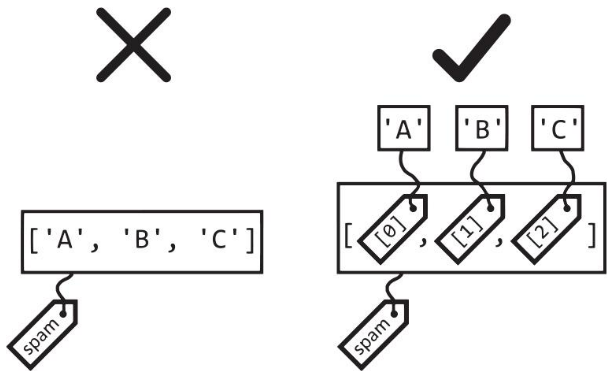

Figure 6-4: Lists don't contain values directly (left); they contain references to values (right).

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

A Short Program: The Matrix Screensaver

In the hacker science fiction film *The Matrix*, computer monitors display streams of glowing green numbers, like digital rain pouring down a glass window. The numbers may be meaningless, but they look cool. Just for fun, we can create our own Matrix screensaver in Python. Enter the following code into a new file and save it as *matrixscreensaver.py*:

```
import random, sys, time

WIDTH = 70  # The number of columns

try:
    # For each column, when the counter is 0,
no stream is shown.
    # Otherwise, it acts as a counter for how
many times a 1 or 0
    # should be displayed in that column.
    columns = [0] * WIDTH
    while True:
```

```
        # Loop over each column:
        for i in range(WIDTH):
            if random.random() < 0.02:
                # Restart a stream counter on
this column.
                # The stream length is
between 4 and 14 characters long.
                columns[i] =
random.randint(4, 14)

            # Print a character in this
column:
            if columns[i] == 0:
                # Change this ' '' to '.' to
see the empty spaces:
                print(' ', end='')
            else:
                # Print a 0 or 1:
                print(random.choice([0, 1]),
end='')
                columns[i] -= 1  # Decrement
the counter for this column.
        print()  # Print a newline at the end
of the row of columns.
        time.sleep(0.1)  # Each row pauses
for one tenth of a second.
except KeyboardInterrupt:
    sys.exit()  # When Ctrl-C is pressed, end
the program.
```

When you run this program, it produces streams of binary 1s and 0s, as in Figure 6-5.

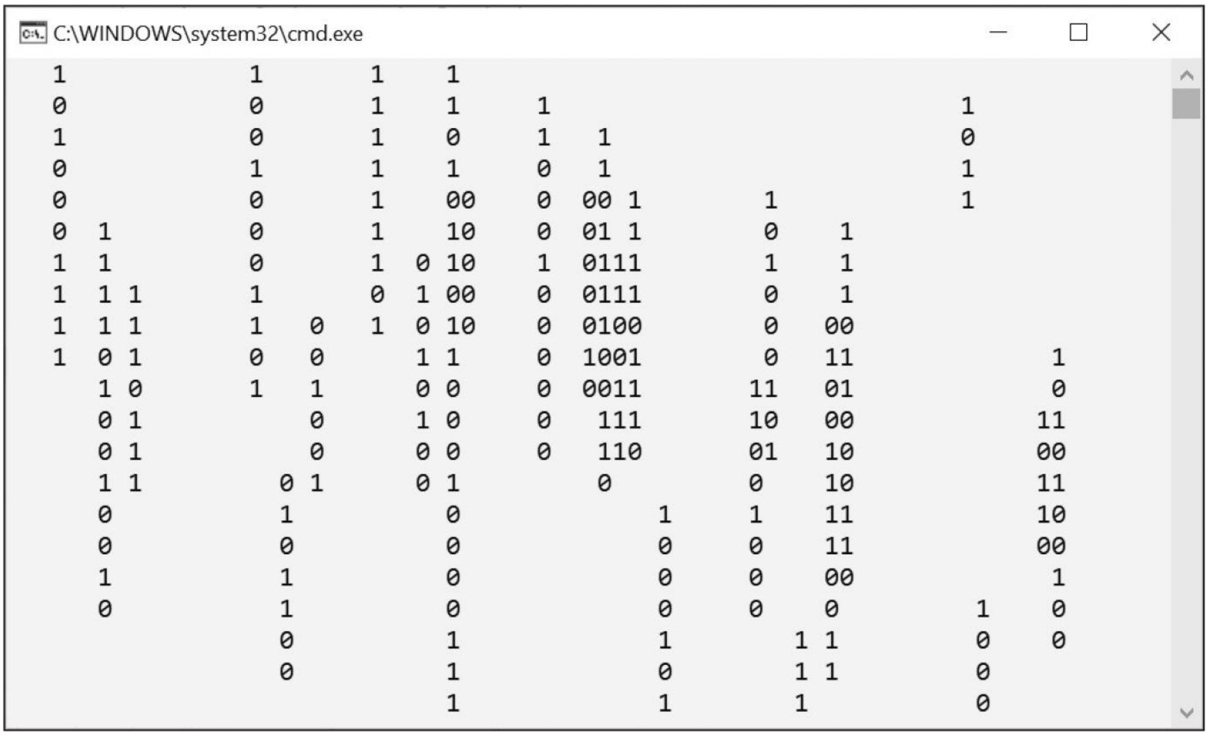

Figure 6-5: The Matrix screensaver program

Like the previous spike and zigzag programs, this program creates a scrolling animation by printing rows of text inside an infinite loop that is stopped when the user presses CTRL-C. The main data structure in this program is the `columns` list, which holds 70 integers, one for each column of output. When an integer in `columns` is `0`, it prints an empty space for that column. When it's greater than `0`, it randomly prints a `0` or `1` and then decrements the integer. Once the integer is reduced to `0`, that column prints an empty space again. The program randomly sets the integers in `columns` to integers between `4` and `14` to produce streams of random binary 0s and 1s.

Let's take a look at each part of the program:

```
import random, sys, time

WIDTH = 70  # The number of columns
```

We import the `random` module for its `choice()` and `randint()` functions, the `sys` module for its `exit()` function, and the `time` module for its `sleep()` function. We also set a variable named `WIDTH` to `70` so that the program produces output for 70 columns of characters. You're free to change this value to a larger or smaller integer based on the size of the window in which you run the program.

The `WIDTH` variable has an all-uppercase name because it's a constant variable. A *constant* is a variable that the code isn't supposed to change once set. Using constants allows you to write more readable code, such as `columns = [0] * WIDTH` instead of `columns = [0] * 70`, which may leave you wondering what the `70` is supposed to be when you reread the code later. In Python, nothing prevents you from changing a constant's value, but the uppercase name can remind the programmer not to do so.

The bulk of the program occurs inside a `try` block, which catches if the user presses CTRL-C to raise a `KeyboardInterrupt` exception:

```
try:
    # For each column, when the counter is 0, 
no stream is shown.
    # Otherwise, it acts as a counter for how 
many times a 1 or 0
    # should be displayed in that column.
    columns = [0] * WIDTH
```

The `columns` variable contains a list of `0` integers. The number of integers in this list is equal to the `WIDTH`. Each of these integers controls whether a column of the output window prints a stream of binary numbers or not:

```
    while True:
        # Loop over each column:
        for i in range(WIDTH):
            if random.random() < 0.02:
                # Restart a stream counter on 
this column.
                # The stream length is 
between 4 and 14 characters long.
                columns[i] = 
random.randint(4, 14)
```

We want this program to run forever, so we place it all inside an infinite `while True:` loop. Inside this loop is a `for` loop that iterates over each column of a single row. The loop variable `i` represents the indexes of columns; it begins at 0 and goes up to but does not include WIDTH. The value in columns[0] represents what should be printed in the leftmost column, columns[1] does so for the second column from the left, and so on.

For each column, there is a two percent chance that the integer at columns[i] is set to a number between 4 and 14. We calculate this chance by comparing random.random() (a function that returns a random float between 0.0 and 1.0) to 0.02. If you want the streams to be denser or sparser, you can increase or decrease this number, respectively. We set the counter integers for each column to a random number between 4 and 14:

```
            # Print a character in this
column:
            if columns[i] == 0:
                # Change this ' '' to '.' to
see the empty spaces:
                print(' ', end='')
            else:
                # Print a 0 or 1:
                print(random.choice([0, 1]),
end='')
                columns[i] -= 1  # Decrement
the counter for this column.
```

Also inside the for loop, the program determines if it should print a random 0 or 1 binary number or an empty space. If columns[i] is 0, it prints an empty space. Otherwise, it passes the list [0, 1] to the random.choice() function, which returns a random value from that list to print. The code also decrements the counter at columns[i] so that it gets closer to 0 and no longer prints binary numbers.

If you'd like to see the "empty" spaces the program prints, try changing the ' ' string to '.' and running the program again. The output should look like this:

```
............................1................
.........................
```

```
................0...........1................
......1..................
................1...........0................
1.....0..................
............1...0...........0.....0..........
1.....0..................
............1.1.1...........0.....0..........
1.....1..1...............
............0.0.0...........0.....1.........0
0.....1..1...............
```

After the `else` block ends, the `for` loop block also ends:

```
        print()  # Print a newline at the end of the row of columns.
        time.sleep(0.1)  # Each row pauses for one tenth of a second.
except KeyboardInterrupt:
    sys.exit()  # When Ctrl-C is pressed, end the program.
```

The `print()` call after the `for` loop prints a newline, as the previous `print()` calls for each column pass the `end=''` keyword argument to prevent a newline from being printed after each column. For each row printed, the program introduces a tenth-of-a-second pause by calling `time.sleep(0.1)`.

The final part of the program is an `except` block that exits the program if the user pressed CTRL-C to raise a `KeyboardInterrupt` exception.

Summary

Lists are useful data types, as they allow you to write code that works on a modifiable number of values in a single variable. Later in this book, you'll see programs that use lists to do things that would otherwise be difficult or impossible.

A list is a sequence data type that is mutable, meaning that its contents can change. Tuples and strings, though also sequence data types, are immutable and cannot be changed. We can overwrite a variable that contains a tuple or string value with a new tuple or string value, which isn't the same thing as modifying the existing value in place—as, say, the `append()` or `remove()` method does on lists.

Variables don't store list values directly; they store references to lists. This is an important distinction when you're 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, assume `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, assume `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 write 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 Programs

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 a *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 heads-tails 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 100 heads and tails. Your program should break up the experiment into two parts: the first part generates a list of 100 randomly selected `'H'` and `'T'` 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 that you can find out what percentage of the coin flips contains a streak of six heads or six tails in a row. As a hint, the function call `random.randint(0, 1)` will return a `0` value 50 percent of the time and a `1` value the other 50 percent of the time.

You can start with the following template:

```
import random
number_of_streaks = 0
for experiment_number in range(10000):  # Run
100,000 experiments total.
    # 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%%' %
(number_of_streaks / 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.

To create a list, use a `for` loop that appends a randomly selected `'H'` or `'T'` to a list 100 times. To determine if there is a streak of six heads or six tails, create a slice like `some_list[i:i + 6]` (which contains the six items starting at index `i`) and then compare it to the list values `['H', 'H', 'H', 'H', 'H', 'H']` and `['T', 'T', 'T', 'T', 'T', 'T']`.