

# Chapter 9

## For Loops

### 9.1 Traversing a string

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to write a traversal is with a `while` statement:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print letter
    index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

**Exercise 9.1.** *Write a function that takes a string as an argument and displays the letters backward, one per line.*

Another way to write a traversal is with a `for` loop:

```
for char in fruit:
    print char
```

Each time through the loop, the next character in the string is assigned to the variable `char`. The loop continues until no characters are left.

The following example shows how to use concatenation (string addition) and a `for` loop to generate an abecedarian series (that is, in alphabetical order). In Robert Mc-

Closkey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'

for letter in prefixes:
    print letter + suffix
```

The output is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Of course, that's not quite right because "Ouack" and "Quack" are misspelled.

**Exercise 9.2.** *Modify the program to fix this error.*

## 9.2 Traversing a list

The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

```
for cheese in cheeses:
    print cheese
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the functions `range` and `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

This loop traverses the list and updates each element. `len` returns the number of elements in the list. `range` returns a list of indices from 0 to  $n - 1$ , where  $n$  is the length of the list. Each time through the loop `i` gets the index of the next element. The assignment statement in the body uses `i` to read the old value of the element and to assign the new value.

A for loop over an empty list never executes the body:

```
for x in empty:
    print 'This never happens.'
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

### 9.3 A find function

What does the following function do?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

This use of loops is the basic logic behind the `find` method discussed earlier.

This is the first example we have seen of a `return` statement inside a loop. If `word[index] == letter`, the function breaks out of the loop and returns immediately.

If the character doesn't appear in the string, the program exits the loop normally and returns `-1`.

This pattern of computation—traversing a sequence and returning when we find what we are looking for—is called a **search**.

**Exercise 9.3.** *Modify `find` so that it has a third parameter, the index in `word` where it should start looking.*

### 9.4 Looping and counting

The following program counts the number of times the letter `a` appears in a string:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print count
```

This program demonstrates another pattern of computation called a **counter**. The variable `count` is initialized to 0 and then incremented each time an `a` is found. When the loop exits, `count` contains the result—the total number of `a`'s.

**Exercise 9.4.** *Encapsulate this code in a function named `count`, and generalize it so that it accepts the string and the letter as arguments.*

**Exercise 9.5.** *Rewrite this function so that instead of traversing the string, it uses the three-parameter version of `find` from the previous section.*

## 9.5 The `in` operator

The operators we have seen so far are all special characters like `+` and `*`, but there are a few operators that are words. `in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second:

```
>>> 'an' in 'banana'
True
>>> 'c' in 'banana'
False
```

For example, the following function prints all the letters from `word1` that also appear in `word2`:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print letter
```

With well-chosen variable names, Python sometimes reads like English. You could read this loop, “for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.”

Here’s what you get if you compare apples and oranges:

```
>>> in_both('apples', 'oranges')
a
e
s
```

The `in` operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

## 9.6 `break`

As with `while` loops, sometimes you don’t know it’s time to end a `for` loop until you get half way through the body. Since we can’t reset a sentinel value to exit the loop, in

this case we use the break statement to jump out of the loop.

For example, suppose you want to count the number of times the word "lemur" appears in a list, but stop early if you see the word "done". You could write:

```
count = 0
animals = ['cat', 'lemur', 'dog', 'fox', 'rabbit', 'lemur', 'done', 'lemur']
for item in animals:
    if item == 'lemur':
        count += 1
    elif item == 'done':
        break
print count
```

The loop now runs until either it consumes all elements of the list or it hits the break statement, and count will be equal to 2 after execution. This way of writing for loops is common because you can check the condition anywhere in the loop (not just at the top).

## 9.7 Map, filter and reduce

To add up all the numbers in a list, you can use a loop like this:

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

total is initialized to 0. Each time through the loop, x gets one element from the list. The += operator provides a short way to update a variable:

```
total += x
```

is equivalent to:

```
total = total + x
```

As the loop executes, total accumulates the sum of the elements; a variable used this way is sometimes called an **accumulator**.

Adding up the elements of a list is such a common operation that Python provides it as a built-in function, sum:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

An operation like this that combines a sequence of elements into a single value is sometimes called **reduce**.

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

`res` is initialized with an empty list; each time through the loop, we append the next element. So `res` is another kind of accumulator.

An operation like `capitalize_all` is sometimes called a **map** because it “maps” a function (in this case the method `capitalize`) onto each of the elements in a sequence.

Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` is a string method that returns `True` if the string contains only upper case letters.

An operation like `only_upper` is called a **filter** because it selects some of the elements and filters out the others.

Most common list operations can be expressed as a combination of `map`, `filter` and `reduce`. Because these operations are so common, Python provides language features to support them, including the built-in function `reduce` and an operator called a “list comprehension.” But these features are idiomatic to Python, so I won’t go into the details.

**Exercise 9.6.** Write a function that takes a list of numbers and returns the cumulative sum; that is, a new list where the  $i$ th element is the sum of the first  $i + 1$  elements from the original list. For example, the cumulative sum of `[1, 2, 3]` is `[1, 3, 6]`.

## 9.8 Debugging

When you are debugging a program, and especially if you are working on a hard bug, there are four things to try:

**reading:** Examine your code, read it back to yourself, and check that it means what you meant to say.

**running:** Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding.

**ruminating:** Take some time to think! What kind of error is it: syntax, run-time, logical? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?

**retreating:** At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works, and that you understand. Then you can start rebuilding.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call "random walk programming," which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

The way out is to take more time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

Taking a break sometimes helps with the thinking. So does talking. If you explain the problem to someone else (or even yourself), you will sometimes find the answer before you finish asking the question.

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that you understand, and that works.

Beginning programmers are often reluctant to retreat, because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can paste the pieces back in a little bit at a time.

To summarize, here's the Seventh Theorem of debugging:

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

**Exercise 9.7.** *Starting with this diagram, execute the program on paper, changing the values of  $i$  and  $j$  during each iteration. Find and fix the second error in this function.*

## 9.9 Glossary

**list:** A sequence of values.

**element:** One of the values in a list (or other sequence), also called items.

**index:** An integer value that indicates an element in a list.

**nested list:** A list that is an element of another list.

**list traversal:** The sequential accessing of each element in a list.

**mapping:** A relationship in which each element of one set corresponds to an element of another set. For example, a list is a mapping from indices to elements.

**accumulator:** A variable used in a loop to add up or accumulate a result.

**reduce:** A processing pattern that traverses a sequence and accumulates the elements into a single result.

**map:** A processing pattern that traverses a sequence and performs an operation on each element.

**filter:** A processing pattern that traverses a list and selects the elements that satisfy some criterion.

**object:** Something a variable can refer to. An object has a type and a value.

**equivalent:** Having the same value.

**identical:** Being the same object (which implies equivalence).

**reference:** The association between a variable and its value.

**aliasing:** A circumstance where two variables refer to the same object.

**delimiter:** A character or string used to indicate where a string should be split.



## 9.10 Exercises

**Exercise 9.8.** *The slice operator can take a third argument that determines the step size, so `t[::2]` creates a list that contains every other element from `t`. If the step size is negative, it goes through the list backward, so `t[::-1]` creates a list of all the elements in `t` in reverse order.*

*Use this idiom to write a one-line version of `is_palindrome` from Exercise 19.7.*

**Exercise 9.9.** *Write a function called `is_sorted` that takes a list as a parameter and returns `True` if the list is sorted in ascending order and `False` otherwise. You can assume (as a precondition) that the elements of the list can be compared with the comparison operators `<`, `>`, etc.*

*For example, `is_sorted([1,2,2])` should return `True` and `is_sorted(['b','a'])` should return `False`.*