Chapter 7

Iteration

7.1 Multiple assignment

As you may have discovered, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```python
bruce = 5
print bruce,
bruce = 7
print bruce
```

The output of this program is 5 7, because the first time `bruce` is printed, its value is 5, and the second time, its value is 7. The comma at the end of the first `print` statement suppresses the newline, which is why both outputs appear on the same line.

Here is what multiple assignment looks like in a state diagram:

```
\begin{tikzpicture}
  \node (bruce) {bruce};
  \node (7) at (1,0) {7};
  \draw[->] (bruce) to (7);
\end{tikzpicture}
```

With multiple assignment it is especially important to distinguish between an assignment operation and a statement of equality. Because Python uses the equal sign (=) for assignment, it is tempting to interpret a statement like `a = b` as a statement of equality. It is not!

First, equality is a symmetric relation and assignment is not. For example, in mathematics, if \( a = 7 \) then \( 7 = a \). But in Python, the statement `a = 7` is legal and \( 7 = a \) is not.
Furthermore, in mathematics, a statement of equality is either true or false, for all time. If \( a = b \) now, then \( a \) will always equal \( b \). In Python, an assignment statement can make two variables equal, but they don’t have to stay that way:

\[
\begin{align*}
   &a = 5 \\
   &b = a \quad \# \text{ a and } b \text{ are now equal} \\
   &a = 3 \quad \# \text{ a and } b \text{ are no longer equal}
\end{align*}
\]

The third line changes the value of \( a \) but does not change the value of \( b \), so they are no longer equal.

Although multiple assignment is frequently helpful, you should use it with caution. If the values of variables change frequently, it can make the code difficult to read and debug.

### 7.2 Updating variables

One of the most common forms of multiple assignment is an **update**, where the new value of the variable depends on the old.

\[
x = x+1
\]

This means “get the current value of \( x \), add one, and then update \( x \) with the new value.”

If you try to update a variable that doesn’t exist, you get an error, because Python evaluates the right side before it assigns a value to \( x \):

\[
>>> x = x+1
\]

```
NameError: name 'x' is not defined
```

Before you can update a variable, you have to **initialize** it, usually with a simple assignment:

\[
>>> x = 0 \\
>>> x = x+1
\]

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**.

### 7.3 The while statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

This repetition is also called **iteration**. Because iteration is so common, Python provides several language features to make it easier. One is the for statement we saw in Section 18.2. We’ll get back to that later.
7.3. The while statement

Another is the while statement. Here is a version of countdown that uses a while statement:

```python
def countdown(n):
    while n > 0:
        print n
        n = n-1
    print 'Blastoff!'  
```

You can almost read the while statement as if it were English. It means, “While n is greater than 0, display the value of n and then reduce the value of n by 1. When you get to 0, display the word Blastoff!”

More formally, here is the flow of execution for a while statement:

1. Evaluate the condition, yielding True or False.
2. If the condition is false, exit the while statement and continue execution at the next statement.
3. If the condition is true, execute the body and then go back to step 1.

This type of flow is called a loop because the third step loops back around to the top.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an infinite loop. An endless source of amusement for computer scientists is the observation that the directions on shampoo, “Lather, rinse, repeat,” are an infinite loop.

In the case of countdown, we can prove that the loop terminates because we know that the value of n is finite, and we can see that the value of n gets smaller each time through the loop, so eventually we have to get to 0. In other cases, it is not so easy to tell:

```python
def sequence(n):
    while n != 1:
        print n,
        if n%2 == 0:
            n = n/2  
        else:
            n = n*3+1
```

The condition for this loop is n != 1, so the loop will continue until n is 1, which makes the condition false.

Each time through the loop, the program outputs the value of n and then checks whether it is even or odd. If it is even, n is divided by 2. If it is odd, the value of n is replaced with n*3+1. For example, if the argument passed to sequence is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.
Since \( n \) sometimes increases and sometimes decreases, there is no obvious proof that \( n \) will ever reach 1, or that the program terminates. For some particular values of \( n \), we can prove termination. For example, if the starting value is a power of two, then the value of \( n \) will be even each time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

The hard question is whether we can prove that this program terminates for all positive values of \( n \). So far, no one has been able to prove it or disprove it!

### 7.4 Sentinel loops

Sometimes you don’t know it’s time to end a loop until you get halfway through the body. In that case you can set a sentinel to watch for a condition and jump out of the loop.

For example, suppose you want to take input from the user until they type done. You could write:

```python
finished = False
while not finished:
    line = raw_input('>')
    if line == 'done':
        finished = True
    else:
        print line

print 'Done!'
```

The loop condition is based on the sentinel `finished`, which begins as `False`, meaning we are not finished with the loop.

Each time through, it prompts the user with an angle bracket. If the user types done, the sentinel activates and will be set to `True`, which exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here’s a sample run:

```
> not done
not done
> done
Done!
```

This way of writing while loops is common because you can check the condition in multiple ways anywhere in the loop (not just at the top).
7.5 Square roots

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, one way of computing square roots is Newton’s method. Suppose that you want to know the square root of \( a \). If you start with almost any estimate, \( x \), you can compute a better estimate with the following formula:

\[
y = \frac{x + a/x}{2}
\]

For example, if \( a \) is 4 and \( x \) is 3:

```python
>>> a = 4.0
>>> x = 3.0
>>> y = (x + a/x) / 2
>>> print y
2.16666666667
```

Which is closer to the correct answer (\( \sqrt{4} = 2 \)). If we repeat the process with the new estimate, it gets even closer:

```python
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00641025641
```

After a few more updates, the estimate is almost exact:

```python
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.000001024003
>>> x = y
>>> x = (x + a/x) / 2
>>> print y
2.00000000003
```

In general we don’t know ahead of time how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing:

```python
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.0
>>> x = y
>>> y = (x + a/x) / 2
```
>>> print y
2.0

When y == x, we can stop. Here is a loop that starts with an initial estimate, x, and improves it until it stops changing:

```python
finished = False
while not finished:
    print x
    y = (x + a/x) / 2
    if y == x:
        finished = True
    x = y
```

For most values of a this works fine, but in general it is dangerous to test float equality. Floating-point values are only approximately right: most rational numbers, like 1/3, and irrational numbers, like $\sqrt{2}$, can’t be represented exactly with a float.

Rather than checking whether x and y are exactly equal, it is safer to use math.fabs to compute the absolute value, or magnitude, of the difference between them:

```python
if math.fabs(y-x) < something_small:
    finished = True
```

Where something_small has a value like 0.0000001 that determines how close is close enough.

**Exercise 7.1.** Wrap this loop in a function called square_root that takes a as a parameter, chooses a reasonable value of x, and returns an estimate of the square root of a.

### 7.6 Debugging

When you use indices to traverse the values in a sequence, it is tricky to get the beginning and end of the traversal right. Here is a function that is supposed to compare two words and return True if one of the words is the reverse of the other, but it contains two errors:

```python
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
```

```python
```
7.6. Debugging

    return False
    i = i+1
    j = j-1

    return True

The first if statement checks whether the words are the same length. If not, we can return False immediately and then, for the rest of the function, we can assume that the words are the same length. This is another example of a guardian.

i and j are indices: i traverses word1 forward while j traverses word2 backward. If we find two letters that don’t match, we can return False immediately. If we get through the whole loop and all the letters match, we return True.

If we test this function with the words “pots” and “stop”, we expect the return value True, but we get an IndexError:

>>> is_reverse('pots', 'stop')
...  
    File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
    IndexError: string index out of range

For debugging this kind of error, my first move is to print the values of the indices immediately before the line where the error appears.

    while j > 0:
        print i, j  # print here

        if word1[i] != word2[j]:
            return False
            i = i+1
            j = j-1

Now when I run the program again, I get more information:

>>> is_reverse('pots', 'stop')
0 4
...
    IndexError: string index out of range

The first time through the loop, the value of j is 4, which is out of range for the string ‘pots’. The index of the last character is 3, so the initial value for j should be len(word2)-1.

If I fix that error and run the program again, I get:

>>> is_reverse('pots', 'stop')
0 3
This time we get the right answer, but it looks like the loop only ran three times, which is suspicious. To get a better idea of what is happening, it is useful to draw a state diagram. During the first iteration, the frame for is_reverse looks like this:

<table>
<thead>
<tr>
<th>word1 --- 'pots'</th>
<th>word2 --- 'stop'</th>
</tr>
</thead>
<tbody>
<tr>
<td>i --- 0</td>
<td>j --- 3</td>
</tr>
</tbody>
</table>

I took a little license by arranging the variables in the frame and adding dotted lines to show that the values of i and j indicate characters in word1 and word2.

### 7.7 Glossary

**multiple assignment**: Making more than one assignment to the same variable during the execution of a program.

**update**: An assignment where the new value of the variable depends on the old.

**initialize**: An assignment that gives an initial value to a variable that will be updated.

**increment**: An update that increases the value of a variable (often by one).

**decrement**: An update that decreases the value of a variable.

**iteration**: Repeated execution of a set of statements using a loop.

**infinite loop**: A loop in which the terminating condition is never satisfied.

### 7.8 Exercises

**Exercise 7.2.** To test the square root algorithm in this chapter, you could compare it with math.sqrt. Write a function named test_square_root that prints a table like this:

```
   1.0  1.0     1.0     0.0
   2.0 1.41421356237 1.41421356237 2.22044604925e-16
   3.0 1.73205080757 1.73205080757 0.0
   4.0 2.0     2.0     0.0
   5.0 2.2360679775 2.2360679775 0.0
   6.0 2.44948974278 2.44948974278 0.0
```
7.8. Exercises

7.0 2.64575131106 2.64575131106 0.0  
8.0 2.82842712475 2.82842712475 4.4408920985e-16  
9.0 3.0 3.0 0.0

The first column is a number, \( a \); the second column is the square root of a computed 
with the function from Exercise 7.1; the third column is the square root computed by 
\texttt{math.sqrt}; the fourth column is the absolute value of the difference between the two 
estimates.

**Exercise 7.3.** The built-in function \texttt{eval} takes a string and evaluates it using the 
Python interpreter. For example:

```python
>>> eval('1 + 2 * 3')
7
>>> import math  
>>> eval('math.sqrt(5)')
2.23606774099998
>>> eval('type(math.pi)')
<type 'float'>
```

Write a function called \texttt{eval_loop} that iteratively prompts the user, takes the resulting 
input and evaluates it using \texttt{eval}, and prints the result.

*It should continue until the user enters `done`, and then return the value of the last 
expression it evaluated.*