# Chapter 2 Writing Simple Programs

## Objectives

- To know the steps in an orderly software development process.
- To understand programs following the Input, Process, Output (IPO) pattern and be able to modify them in simple ways.
- To understand the rules for forming valid Python identifiers and expressions.
- To be able to understand and write Python statements to output information to the screen, assign values to variables, get numeric information entered from the keyboard, and perform a counted loop

## 2.1 The Software Development Process

As you saw in the previous chapter, it is easy to run programs that have already been written. The hard part is actually coming up with the program in the first place. Computers are very literal, and they must be told what to do right down to the last detail. Writing large programs is a daunting challenge. It would be almost impossible without a systematic approach.

The process of creating a program is often broken down into stages according to the information that is produced in each phase. In a nutshell, here's what you should do:

**Analyze the Problem** Figure out exactly what the problem to be solved is. Try to understand as much as possible about it. Until you really know what the problem is, you cannot begin to solve it.

**Determine Specifications** Describe exactly what your program will do. At this point, you should not worry about *how* your program will work, but rather about deciding exactly *what* it will accomplish. For simple programs this involves carefully describing what the inputs and outputs of the program will be and how they relate to each other.

**Create a Design** Formulate the overall structure of the program. This is where the *how* of the program gets worked out. The main task is to design the algorithm(s) that will meet the specifications.

**Implement the Design** Translate the design into a computer language and put it into the computer. In this book, we will be implementing our algorithms as Python programs.

**Test/Debug the Program** Try out your program and see if it works as expected. If there are any errors (often called *bugs*), then you should go back and fix them. The process of locating and fixing errors is called *debugging* a program. During the debugging phase, your goal is to find errors, so you should try everything you can think of that might "break" the program. It's good to keep in mind the old maxim: "Nothing is foolproof because fools are too ingenious."

**Maintain the Program** Continue developing the program in response to the needs of your users. Most programs are never really finished; they keep evolving over years of use.

## 2.2 Example Program: Temperature Converter

Let's go through the steps of the software development process with a simple real-world example involving a fictional computer science student, Susan Computewell.

Susan is spending a year studying in Germany. She has no problems with language, as she is fluent in many languages (including Python). Her problem is that she has a hard time figuring out the temperature in the morning so that she knows how to dress for the day. Susan listens to the weather report each morning, but the temperatures are given in degrees Celsius, and she is used to Fahrenheit.

Fortunately, Susan has an idea to solve the problem. Being a computer science major, she never goes anywhere without her laptop computer. She thinks it might be possible that a computer program could help her out.

Susan begins with an analysis of her problem. In this case, the problem is pretty clear: the radio announcer gives temperatures in degrees Celsius, but Susan only comprehends temperatures that are in degrees Fahrenheit.

Next, Susan considers the specifications of a program that might help her out. What should the input be? She decides that her program will allow her to type in the temperature in degrees Celsius. And the output? The program will display the temperature converted into degrees Fahrenheit. Now she needs to specify the exact relationship of the output to the input.

Susan does some quick figuring. She knows that 0 degrees Celsius (freezing) is equal to 32 degrees Fahrenheit, and 100 Celsius (boiling) is equal to 212 Fahrenheit. With this information, she computes the ratio of Fahrenheit to Celsius degrees as $\frac{212-32}{100-0} = \frac{180}{100} = \frac{9}{5}$. Using F to represent the Fahrenheit temperature and C for Celsius, the conversion formula will have the form $F = \frac{9}{5}C + k$ for some constant $k$. Plugging in 0 and 32 for $C$ and $F$, respectively, Susan immediately sees that $k = 32$. So, the final formula for the relationship is $F = \frac{9}{5}C + 32$. That seems an adequate specification.

Notice that this describes one of many possible programs that could solve this problem. If Susan had background in the field of Artificial Intelligence (AI), she might consider writing a program that would actually listen to the radio announcer to get the current temperature using speech recognition algorithms. For output, she might have the computer control a robot that goes to her closet and picks an appropriate outfit based on the converted temperature. This would be a much more ambitious project, to say the least!

Certainly, the robot program would also solve the problem identified in the problem analysis. The purpose of specification is to decide exactly what this particular program will do to solve a problem. Susan knows better than to just dive in and start writing a program without first having a clear idea of what she is trying to build.

Susan is now ready to design an algorithm for her problem. She immediately realizes that this is a simple algorithm that follows a standard pattern: *Input, Process, Output* (IPO). Her program will prompt the user for some input information (the Celsius temperature), process it to convert to a Fahrenheit temperature, and then output the result by displaying it on the computer screen.

Susan could write her algorithm down in a computer language. However, the precision of writing it out formally tends to stifle the creative process of developing the algorithm. Instead, she writes her algorithm using *pseudocode*. Pseudocode is just precise English that describes what a program does. It is meant to communicate algorithms without all the extra mental overhead of getting the details right in any particular programming language.

Here is Susan's completed algorithm:

```
Input the temperature in degrees Celsius (call it celsius)
Calculate fahrenheit as (9/5)celsius + 32
Output fahrenheit
```

The next step is to translate this design into a Python program. This is straightforward, as each line of the algorithm turns into a corresponding line of Python code.

```
# convert.py
#     A program to convert Celsius temps to Fahrenheit
# by: Susan Computewell

def main():
    celsius = input("What is the Celsius temperature? ")
    fahrenheit = (9.0 / 5.0) * celsius + 32
    print "The temperature is", fahrenheit, "degrees Fahrenheit."

main()
```

See if you can figure out what each line of this program does. Don't worry if some parts are a bit confusing. They will be discussed in detail in the next section.

After completing her program, Susan tests it to see how well it works. She uses inputs for which she knows the correct answers. Here is the output from two of her tests:

```
What is the Celsius temperature? 0
The temperature is 32.0 degrees fahrenheit.

What is the Celsius temperature? 100
The temperature is 212.0 degrees fahrenheit.
```

You can see that Susan used the values of 0 and 100 to test her program. It looks pretty good, and she is satisfied with her solution. She is especially pleased that no debugging seems necessary (which is very unusual).

## 2.3 Elements of Programs

Now that you know something about the programming process, you are *almost* ready to start writing programs on your own. Before doing that, though, you need a more complete grounding in the fundamentals of Python. The next few sections will discuss technical details that are essential to writing correct programs. This material can seem a bit tedious, but you will have to master these basics before plunging into more interesting waters.

### 2.3.1 Names

You have already seen that names are an important part of programming. We give names to modules (e.g., `convert`) and to the functions within modules (e.g., `main`). Variables are used to give names to values (e.g., `celsius` and `fahrenheit`). Technically, all these names are called *identifiers*. Python has some rules about how identifiers are formed. Every identifier must begin with a letter or underscore (the "_" character) which may be followed by any sequence of letters, digits, or underscores. This implies that a single identifier cannot contain any spaces.

According to these rules, all of the following are legal names in Python:

```
x
celsius
spam
spam2
SpamAndEggs
Spam_and_Eggs
```

Identifiers are case-sensitive, so `spam`, `Spam`, `sPam`, and `SPAM` are all different names to Python. For the most part, programmers are free to choose any name that conforms to these rules. Good programmers always try to choose names that describe the thing being named.

One other important thing to be aware of is that some identifiers are part of Python itself. These names are called *reserved words* and cannot be used as ordinary identifiers. The complete list of Python reserved words is shown in Table 2.1.

| | | | | |
|---|---|---|---|---|
| and | del | for | is | raise |
| assert | elif | from | lambda | return |
| break | else | global | not | try |
| class | except | if | or | while |
| continue | exec | import | pass | yield |
| def | finally | in | print | |

Table 2.1: Python Reserved Words.

## 2.3.2 Expressions

Programs manipulate data. The fragments of code that produce or calculate new data values are called *expressions*. So far our program examples have dealt mostly with numbers, so I'll use numeric data to illustrate expressions.

The simplest kind of expression is a *literal*. A literal is used to indicate a specific value. In `chaos.py` you can find the numbers `3.9` and `1`. The `convert.py` program contains `9.0`, `5.0`, and `32`. These are all examples of numeric literals, and their meaning is obvious: `32` represents, well, 32.

A simple identifier can also be an expression. We use identifiers as variables to give names to values. When an identifier appears in an expression, this value is retrieved to provide a result for the expression. Here is an interaction with the Python interpreter that illustrates the use of variables as expressions:

```
>>> x = 5
>>> x
5
>>> print x
5
>>> print spam
Traceback (innermost last):
  File "<pyshell#34>", line 1, in ?
    print spam
NameError: spam
>>>
```

First the variable x is assigned the value 5 (using the numeric literal 5). The next line has Python evaluate the expression x. Python spits back 5, which is the value that was just assigned to x. Of course, we get the same result when we put x in a print statement. The last example shows what happens when we use a variable that has not been assigned a value. Python cannot find a value, so it reports a *Name Error*. This says that there is no value with that name. A variable must always be assigned a value before it can be used in an expression.

More complex and interesting expressions can be constructed by combining simpler expressions with *operators*. For numbers, Python provides the normal set of mathematical operations: addition, subtraction, multiplication, division, and exponentiation. The corresponding Python operators are: +, -, *, /, and **. Here are some examples of complex expressions from chaos.py and convert.py

```
3.9 * x * (1 - x)
9.0 / 5.0 * celsius + 32
```

Spaces are irrelevant within an expression. The last expression could have been written 9.0/5.0*celsius+32 and the result would be exactly the same. Usually it's a good idea to place some spaces in expressions to make them easier to read.

Python's mathematical operators obey the same rules of precedence and associativity that you learned in your math classes, including using parentheses to modify the order of evaluation. You should have little trouble constructing complex expressions in your own programs. Do keep in mind that only the round parentheses are allowed in expressions, but you can nest them if necessary to create expressions like this.

```
((x1 - x2) / 2*n) + (spam / k**3)
```

If you are reading carefully, you may be curious why, in her temperature conversion program, Susan chose to write 9.0/5.0 rather than 9/5. Both of these are legal expressions, but they give different results. This mystery will be discussed in Chapter 3. If you can't stand the wait, try them out for yourself and see if you can figure out what's going on.

## 2.4 Output Statements

Now that you have the basic building blocks, identifier and expression, you are ready for a more complete description of various Python statements. You already know that you can display information on the screen using Python's print statement. But what exactly can be printed? Python, like all programming languages, has a precise set of rules for the syntax (form) and semantics (meaning) of each statement. Computer scientists have developed sophisticated notations called *meta-languages* for describing programming languages. In this book we will rely on a simple template notation to illustrate the syntax of statements.

Here are the possible forms of the `print` statement:

```
print
print <expr>
print <expr>, <expr>, ..., <expr>
print <expr>, <expr>, ..., <expr>,
```

In a nutshell, these templates show that a `print` statement consists of the keyword `print` followed by zero or more expressions, which are separated by commas. The angle bracket notation (<>) is used to indicate "slots" that are filled in by other fragments of Python code. The name inside the brackets indicates what is missing; `expr` stands for an expression. The ellipses ("...") indicate an indefinite series (of expressions, in this case). You don't actually type the dots. The fourth version shows that a `print` statement may be optionally ended with a comma. That is all there is to know about the syntax of `print`.

As far as semantics are concerned, a `print` statement displays information in textual form. Any supplied expressions are evaluated left to right, and the resulting values are displayed on a single line of output in a left-to-right fashion. A single blank space character is placed between the displayed values.

Normally, successive `print` statements will display on separate lines of the screen. A bare `print` (first version above) can be used to get a blank line of output. If a `print` statement ends with a comma (fourth version), a final space is appended to the line, but the output does not advance to the next line. Using this method, multiple `print` statements can be used to generate a single line of output.

Putting it all together, this sequence of `print` statements

```
print 3+4
print 3, 4, 3 + 4
print
print 3, 4,
print 3+4
print "The answer is", 3 + 4
```

produces this output:

```
7
3 4 7

3 4 7
The answer is 7
```

That last `print` statement may be a bit confusing. According to the syntax templates above, `print` requires a sequence of expressions. That means `"The answer is"` must be an expression. In fact, it *is* an expression, but it doesn't produce a number. Instead, it produces another kind of data called a *string*. A sequence of characters enclosed in quotes is a string literal. Strings will be discussed in detail in a later chapter. For now, consider this a convenient way of labeling output.

## 2.5 Assignment Statements

### 2.5.1 Simple Assignment

One of the most important kinds of statements in Python is the assignment statement. We've already seen a number of these in our previous examples. The basic assignment statement has this form:

```
<variable> = <expr>
```

Here `variable` is an identifier and `expr` is an expression. The semantics of the assignment is that the expression on the right side is evaluated to produce a value, which is then associated with the variable named on the left side.

Here are some of the assignments we've already seen:

```
x = 3.9 * x * (1 - x)
fahrenheit = 9.0 / 5.0 * celsius + 32
x = 5
```

A variable can be assigned many times. It always retains the value of the most recent assignment. Here is an interactive Python session that demonstrates the point:

```
>>> myVar = 0
>>> myVar
0
```

```
>>> myVar = 7
>>> myVar
7
>>> myVar = myVar + 1
>>> myVar
8
```

The last assignment statement shows how the current value of a variable can be used to update its value. In this case I simply added one to the previous value. The `chaos.py` program from Chapter 1 did something similar, though a bit more complex. Remember, the values of variables can change; that's why they're called variables.

Sometimes it's helpful to think of a variable as a sort of named storage location in computer memory, a box that we can put a value in. When the variable changes, the old value is erased and a new one written in. Figure 2.1 shows how we might picture the effect of `x = x + 1` using this model. This is exactly the way assignment works in some computer languages. It's also a very simple way to view the effect of assignment, and you'll find pictures similar to this throughout the book.

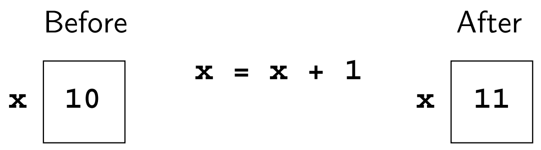

Figure 2.1: Variable as box view of `x = x + 1`

Python assignment statements are actually slightly different from the "variable as a box" model. In Python, values may end up anywhere in memory, and variables are used to refer to them. Assigning a variable is like putting one of those little yellow sticky notes on the value and saying, "this is x." Figure 2.2 gives a more accurate picture of the effect of assignment in Python. An arrow is used to show which value a variable refers to. Notice that the old value doesn't get erased by the new one; the variable simply switches to refer to the new value. The effect is like moving the sticky note from one object to another. This is the way assignment actually works in Python, so you'll see some of these sticky-note style pictures sprinkled throughout the book as well.

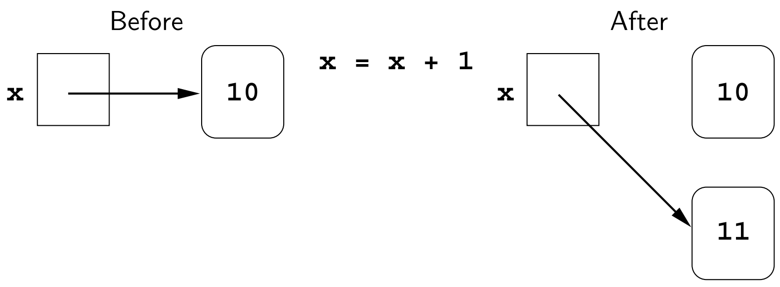

Figure 2.2: Variable as sticky note (Python) view of `x = x + 1`

By the way, even though the assignment statement doesn't directly cause the old value of a variable to be erased and overwritten, you don't have to worry about computer memory getting filled up with the "discarded" values. When a value is no longer referred to by *any* variable, it is no longer useful. Python will automatically clear these values out of memory so that the space can be used for new values. This is like going through your closet and tossing out anything that doesn't have a sticky note to label it. In fact, this process of automatic memory management is actually called *garbage collection*.

## 2.5.2 Assigning Input

The purpose of an input statement is to get some information from the user of a program and store it into a variable. Some programming languages have a special statement to do this. In Python, input is accomplished using an assignment statement combined with a special expression called `input`. This template shows the standard form.

```
<variable> = input(<prompt>)
```

Here `prompt` is an expression that serves to prompt the user for input; this is almost always a string literal (i.e., some text inside of quotation marks).

When Python encounters an `input` expression, it evaluates the prompt and displays the result of the prompt on the screen. Python then pauses and waits for the user to type an expression and press the <`Enter`> key. The expression typed by the user is then evaluated to produce the result of the `input`. This sounds complicated, but most uses of `input` are straightforward. In our example programs, `input` statements are used to get numbers from the user.

```
x = input("Please enter a number between 0 and 1: ")
celsius = input("What is the Celsius temperature? ")
```

If you are reading programs carefully, you probably noticed the blank space inside the quotes at the end of these prompts. I usually put a space at the end of a prompt so that the input that the user types does not start right next to the prompt. Putting a space in makes the interaction easier to read and understand.

Although these two examples specifically prompt the user to enter a number, a number is just a numeric literal—a simple Python expression. In fact, any valid expression would be just as acceptable. Consider the following interaction with the Python interpreter:

```
>>> ans = input("Enter an expression: ")
Enter an expression: 3 + 4 * 5
>>> print ans
23
>>>
```

Here, when prompted to enter an expression, the user typed "3 + 4 * 5." Python evaluated this expression and stored the value in the variable `ans`. When printed, we see that `ans` got the value 23 as expected.

In a way, the `input` is like a delayed expression. The example interaction produced exactly the same result as if we had simply done `ans = 3 + 4 * 5`. The difference is that the expression was supplied at the time the statement was executed instead of being determined when the statement was written by the programmer. Thus, the user can supply formulas for a program to evaluate.

### 2.5.3 Simultaneous Assignment

There is an alternative form of the assignment statement that allows us to calculate several values all at the same time. It looks like this:

```
<var>, <var>, ..., <var> = <expr>, <expr>, ..., <expr>
```

This is called *simultaneous assignment*. Semantically, this tells Python to evaluate all the expressions on the right-hand side and then assign these values to the corresponding variables named on the left-hand side. Here's an example:

```
sum, diff = x+y, x-y
```

Here sum would get the sum of x and y and diff would get the difference.

This form of assignment seems strange at first, but it can prove remarkably useful. Here's an example: Suppose you have two variables x and y and you want to swap the values. That is, you want the value currently stored in x to be in y and the value that is currently in y to be stored in x. At first, you might think this could be done with two simple assignments.

```
x = y
y = x
```

This doesn't work. We can trace the execution of these statements step-by-step to see why.

Suppose x and y start with the values 2 and 4. Let's examine the logic of the program to see how the variables change. The following sequence uses comments to describe what happens to the variables as these two statements are executed:

```
# variables       x  y
# initial values  2  4
x = y
# now             4  4
y = x
# final           4  4
```

See how the first statement clobbers the original value of x by assigning to it the value of y? When we then assign x to y in the second step, we just end up with two copies of the original y value.

One way to make the swap work is to introduce an additional variable that temporarily remembers the original value of x.

```
temp = x
x = y
y = temp
```

Let's walk-through this sequence to see how it works.

```
# variables       x  y  temp
# initial values  2  4  no value yet
temp = x
#                 2  4   2
x = y
```

```
#                  4  4   2
y = temp
#                  4  2   2
```

As you can see from the final values of x and y, the swap was successful in this case.

This sort of three-way shuffle is common in other programming languages. In Python, the simultaneous assignment statement offers an elegant alternative. Here is a simpler Python equivalent:

```
x, y = y, x
```

Because the assignment is simultaneous, it avoids wiping out one of the original values.

Simultaneous assignment can also be used to get multiple values from the user in a single input. Consider this program for averaging exam scores:

```
# avg2.py
#   A simple program to average two exam scores
#   Illustrates use of multiple input

def main():
    print "This program computes the average of two exam scores."

    score1, score2 = input("Enter two scores separated by a comma: ")
    average = (score1 + score2) / 2.0

    print "The average of the scores is:", average

main()
```

The program prompts for two scores separated by a comma. Suppose the user types 86, 92. The effect of the input statement is then the same as if we had done this assignment:

```
score1, score2 = 86, 92
```

We have gotten a value for each of the variables in one fell swoop. This example used just two values, but it could be generalized to any number of inputs.

Of course, we could have just gotten the input from the user using separate input statements.

```
score1 = input("Enter the first score: ")
score2 = input("Enter the second score: ")
```

In some ways this may be better, as the separate prompts are more informative for the user. In this example the decision as to which approach to take is largely a matter of taste. Sometimes getting multiple values in a single `input` provides a more intuitive user interface, so it's a nice technique to have in your toolkit.

## 2.6 Definite Loops

You already know that programmers use loops to execute a sequence of statements several times in succession. The simplest kind of loop is called a *definite loop*. This is a loop that will execute a definite number of times. That is, at the point in the program when the loop begins, Python knows how many times to go around (or *iterate*) the body of the loop. For example, the Chaos program from Chapter 1 used a loop that always executed exactly ten times.

```
for i in range(10):
    x = 3.9 * x * (1 - x)
    print x
```

This particular loop pattern is called a *counted loop*, and it is built using a Python `for` statement. Before considering this example in detail, let's take a look at what `for` loops are all about.

A Python `for` loop has this general form:

```
for <var> in <sequence>:
    <body>
```

The body of the loop can be any sequence of Python statements. The start and end of the body is indicated by its indentation under the loop heading (the `for <var> in <sequence>:` part).

The variable after the keyword `for` is called the *loop index*. It takes on each successive value in the `sequence`, and the statements in the `body` are executed once for each value. Often the `sequence` portion consists of a *list* of values. Lists are a very important concept in Python, and you will learn more about them in upcoming chapters. For now, it's enough to know that you can create a simple list by placing a sequence of expressions in square brackets. Some interactive examples help to illustrate the point:

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

0
1
2
3

>>> for odd in [1, 3, 5, 7, 9]:
        print odd * odd

1
9
25
49
81
```

You can see what is happening in these two examples. The body of the loop is executed using each successive value in the list. The length of the list determines the number of times the loop will execute. In the first example, the list contains the four values 0 through 3, and these successive values of `i` are simply printed. In the second example, `odd` takes on the values of the first five odd natural numbers, and the body of the loop prints the squares of these numbers.

Now, let's go back to the example which began this section (from `chaos.py`) Look again at the loop heading:

```
for i in range(10):
```

Comparing this to the template for the `for` loop shows that the last portion, `range(10)`, must be some kind of sequence. Let's see what the Python interpreter tells us.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Do you see what is happening here? The `range` function is a built-in Python command that simply produces a list of numbers. The loop using `range(10)` is exactly equivalent to one using a list of 10 numbers.

```
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
```

In general, `range(<expr>)` will produce a list of numbers that starts with 0 and goes up to, but not does not include, the value of `<expr>`. If you think about it, you will see that the value of the expression determines the number of items in the resulting list. In `chaos.py` we did not even care what values the loop index variable used (since the value of `i` was not referred to anywhere in the loop body). We just needed a list length of 10 to make the body execute 10 times.

As I mentioned above, this pattern is called a *counted loop*, and it is a very common way to use definite loops. When you want to do something in your program a certain number of times, use a `for` loop with a suitable `range`.

```
for <variable> in range(<expr>):
```

The value of the expression determines how many times the loop executes. The name of the index variable doesn't really matter much; programmers often use `i` or `j` as the loop index variable for counted loops. Just be sure to use an identifier that you are not using for any other purpose. Otherwise you might accidentally wipe out a value that you will need later.

The interesting and useful thing about loops is the way that they alter the "flow of control" in a program. Usually we think of computers as executing a series of instructions in strict sequence. Introducing a loop causes Python to go back and do some statements over and over again. Statements like the `for` loop are called *control structures* because they control the execution of other parts of the program.

Some programmers find it helpful to think of control structures in terms of pictures called *flowcharts*. A flowchart is a diagram that uses boxes to represent different parts of a program and arrows between the boxes to show the sequence of events when the program is running. Figure 2.3 depicts the semantics of the `for` loop as a flowchart.

If you are having trouble understanding the `for` loop, you might find it useful to study the flowchart. The diamond shape box in the flowchart represents a decision in the program. When Python gets to the loop heading, it checks to see if there are any (more) items left in the sequence. If the answer is yes, the loop index variable is assigned the next item in the sequence, and then the loop body is executed. Once the body is complete, the program goes back to the loop heading and checks for another value in the sequence. The loop quits when there are no more items, and the program moves on to the statements that come after the loop.

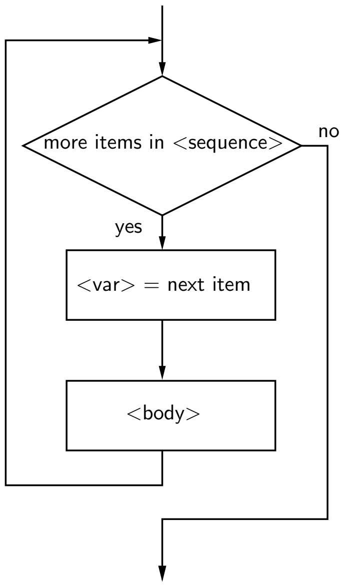

Figure 2.3: Flowchart of a `for` loop.

## 2.7 Example Program: Future Value

Let's close the chapter with one more example of the programming process in action. We want to develop a program to determine the future value of an investment. We'll start with an analysis of the problem. You know that money deposited in a bank account earns interest, and this interest accumulates as the years pass. How much will an account be worth ten years from now? Obviously, it depends on how much money we start with (the principal) and how much interest the account earns. Given the principal and the interest rate, a program should be able to calculate the value of the investment ten years into the future.

We continue by developing the exact specifications for the program. Remember, this is a description of what the program will do. What exactly should the inputs be? We need the user to enter the initial amount to invest, the principal. We will also need some indication of how much interest the account earns. This depends both on the interest rate and how often the interest is compounded. One simple way of handling this is to have the user enter an annual percentage rate. Whatever the actual interest rate and compounding frequency, the annual rate tells us how much the investment accrues in one year. If the annual interest is 3%, then a $100 investment will grow to $103 in one year's time. How should the user represent an annual rate of 3%? There are a number of reasonable choices. Let's assume the user supplies a decimal, so the rate would be entered as 0.03.

This leads us to the following specification:

**Program** Future Value

**Inputs**

> **principal** The amount of money being invested in dollars.
>
> **apr** The annual percentage rate expressed as a decimal number.

**Output** The value of the investment 10 years into the future.

**Relationship** Value after one year is given by $principal(1 + apr)$. This formula needs to be applied 10 times.

Next we design an algorithm for the program. We'll use pseudocode, so that we can formulate our ideas without worrying about all the rules of Python. Given our specification, the algorithm seems straightforward.

```
Print an introduction
Input the amount of the principal (principal)
Input the annual percentage rate (apr)
Repeat 10 times:
    principal = principal * (1 + apr)
Output the value of principal
```

If you know a little bit about financial math (or just some basic algebra), you probably realize that the loop in this design is not strictly necessary; there is a formula for calculating future value in a single step using exponentiation. I have used a loop here both to illustrate another counted loop, and also because this version will lend itself to some modifications that are discussed in the programming exercises at the end of the chapter. In any case, this design illustrates that sometimes an algorithmic approach to a calculation can make the mathematics easier. Knowing how to calculate the interest for just one year allows us to calculate any number of years into the future.

Now that we've thought the problem all the way through in pseudocode, it's time to put our new Python knowledge to work and develop a program. Each line of the algorithm translates into a statement of Python.

Print an introduction (`print` statement, Section 2.4)
```
print "This program calculates the future value"
print "of a 10-year investment."
```

Input the amount of the principal (`input` statement, Section 2.5.2)
```
principal = input("Enter the initial principal:  ")
```

Input the annual percentage rate (`input` expression, Section 2.5.2)
```
apr = input("Enter the annual interest rate:  ")
```

Repeat 10 times: (counted loop, Section 2.6)
```
for i in range(10):
```

Calculate principal = principal * (1 + apr) (simple assignment, Section 2.5.1)
```
    principal = principal * (1 + apr)
```

Output the value of the principal (`print` statement, Section 2.4)
```
print "The value in 10 years is:", principal
```

All of the statement types in this program have been discussed in detail in this chapter. If you have any questions, you should go back and review the relevant descriptions. Notice especially the counted loop pattern is used to apply the interest formula 10 times.

That about wraps it up. Here is the completed program:

```
# futval.py
#    A program to compute the value of an investment
#    carried 10 years into the future

def main():
    print "This program calculates the future value
    print "of a 10-year investment."

    principal = input("Enter the initial principal: ")
    apr = input("Enter the annual interest rate: ")

    for i in range(10):
        principal = principal * (1 + apr)
    print "The value in 10 years is:", principal

main()
```

Notice that I have added a few blank lines to separate the input, processing, and output portions of the program. Strategically placed "white space" can help make your programs more readable.

That's about it for this example; I leave the testing and debugging as an exercise for you.

## 2.8 Chapter Summary

This chapter has covered a lot of ground laying out both the process that is used to develop programs and the details of Python that are necessary to implement simple programs. Here is a quick summary of some of the key points:

- Writing programs requires a systematic approach to problem solving and involves the following steps:
  1. Problem Analysis: Studying the problem to be solved.
  2. Program Specification: Deciding exactly what the program will do.
  3. Design: Writing an algorithm in pseudocode.
  4. Implementation: Translating the design into a programming language.
  5. Testing/Debugging: Finding and fixing errors in the program.
  6. Maintenance: Keeping the program up to date with evolving needs.
- Many simple programs follow the input, process, output (IPO) pattern.
- Programs are composed of statements that are built from identifiers and expressions.
- Identifiers are names; they begin with an underscore or letter which can be followed by a combination of letter, digit, or underscore characters. Identifiers in Python are case sensitive.
- Expressions are the fragments of a program that produce data. An expression can be composed of the following components:

**literals** A literal is a representation of a specific value. For example `3` is a literal representing the number three.

**variables** A variable is an identifier that stores a value.

**operators** Operators are used to combine expressions into more complex expressions. For example, in `x + 3 * y` the operators `+` and `*` are used.

- The Python operators for numbers include the usual arithmetic operations of addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`**`).
- The Python output statement `print` displays the values of a series of expressions to the screen.
- In Python, assignment of a value to a variable is done using the equal sign (`=`). Using assignment, programs can get input from the keyboard. Python also allows simultaneous assignment, which is useful for getting multiple input values with a single prompt.
- Definite loops are loops that execute a known number of times. The Python `for` statement is a definite loop that iterates through a sequence of values. A Python list is often used in a `for` loop to provide a sequence of values for the loop.
- One important use of a `for` statement is in implementing a counted loop, which is a loop designed specifically for the purpose of repeating some portion of the program a specific number of times. A counted loop in Python is created by using the built-in `range` function to produce a suitably sized list of numbers.

## 2.9 Exercises

### Review Questions

#### True/False

1. The best way to write a program is to immediately write down some code and then debug it until it works.
2. An algorithm can be written without using a programming language.

3. Programs no longer require modification after they are written and debugged.
4. Python identifiers must start with a letter or underscore.
5. Reserved words make good variable names.
6. Expressions are built from literals, variables, and operators.
7. In Python, `x = x + 1` is a legal statement.
8. Python does not allow the input of multiple values with a single statement.
9. A counted loop is designed to iterate a specific number of times.
10. In a flowchart, diamonds are used to show statements, and rectangles are used for decision points.

**Multiple Choice**

1. Which of the following is *not* a step in the software development process?
   a) Specification b) Testing/Debugging
   c) Fee setting d) Maintenance
2. What is the correct formula for converting Celsius to Fahrenheit?
   a) $F = 9/5(C) + 32$ b) $F = 5/9(C) - 32$
   c) $F = B^2 - 4AC$ d) $F = \frac{212-32}{100-0}$
3. The process of describing exactly *what* a computer program will do to solve a problem is called
   a) design b) implementation c) programming d) specification
4. Which of the following is *not* a legal identifier?
   a) `spam` b) `spAm` c) `2spam` d) `spam4U`
5. Which of the following are *not* used in expressions?
   a) variables b) statements c) operators d) literals
6. Fragments of code that produce or calculate new data values are called
   a) identifiers b) expressions
   c) productive clauses d) assignment statements

7. Which of the following is *not* a part of the IPO pattern?
   a) Input  b) Program  c) Process  d) Output

8. The template `for <variable> in range(<expr>)` describes
   a) a general for loop  b) an assignment statement
   c) a flowchart  d) a counted loop

9. Which of the following is the most accurate model of assignment in Python?
   a) sticky-note  b) variable-as-box
   c) simultaneous  d) plastic-scale

10. In Python, getting user input is done with a special expression called
    a) `for`  b) `read`  c) simultaneous assignment  d) `input`

## Discussion

1. List and describe in your own words the six steps in the software development process.

2. Write out the `chaos.py` program (Section 1.6) and identify the parts of the program as follows:

   - Circle each identifier.
   - Underline each expression.
   - Put a comment at the end of each line indicating the type of statement on that line (output, assignment, input, loop, etc.)

3. Explain the relationships among the concepts: definite loop, `for` loop, and counted loop.

4. Show the output from the following fragments:

   (a)
   ```
   for i in range(5):
       print i * i
   ```
   (b)
   ```
   for d in [3,1,4,1,5]:
       print d,
   ```
   (c)
   ```
   for i in range(4):
       print "Hello"
   ```
   (d)
   ```
   for i in range(5):
       print i, 2**i
   ```

5. Why is it a good idea to first write out an algorithm in pseudocode rather than jumping immediately to Python code?

## Programming Exercises

1. A user-friendly program should print an introduction that tells the user what the program does. Modify the `convert.py` program (Section 2.2) to print an introduction.
2. Modify the `avg2.py` program (Section 2.5.3) to find the average of three exam scores.
3. Modify the `convert.py` program (Section 2.2) with a loop so that it executes 5 times before quitting (i.e., it converts 5 temperatures in a row).
4. Modify the `convert.py` program (Section 2.2) so that it computes and prints a table of Celsius temperatures and the Fahrenheit equivalents every 10 degrees from 0C to 100C.
5. Modify the `futval.py` program (Section 2.7) so that the number of years for the investment is also a user input. Make sure to change the final message to reflect the correct number of years.
6. Suppose you have an investment plan where you invest a certain fixed amount every year. Modify `futval.py` to compute the total accumulation of your investment. The inputs to the program will be the amount to invest each year, the interest rate, and the number of years for the investment.
7. As an alternative to APR, the interest accrued on an account is often described in terms of a nominal rate and the number of compounding periods. For example, if the interest rate is 3% and the interest is compounded quarterly, the account actually earns 3/4 % interest every 3 months.

   Modify the `futval.py` program to use this method of entering the interest rate. The program should prompt the user for the yearly rate (`rate`) and the number of times that the interest is compounded each year (`periods`). To compute the value in ten years, the program will loop `10 * periods` times and accrue `rate/period` interest on each iteration.
8. Write a program that converts temperatures from Fahrenheit to Celsius.
9. Write a program that converts distances measured in kilometers to miles. One kilometer is approximately 0.62 miles.

10. Write a program to perform a unit conversion of your own choosing. Make sure that the program prints an introduction that explains what it does.