Chapter 7

COMPONENTS OF PROGRAMMING

The idea of "writing a computer program" can sound like a difficult and complicated job, full of strange terminology and complex rules. There are many elements, though, that are universal to nearly all programming languages. Some of these common building blocks are the subject of this chapter. By understanding these components in general terms, you will be better able to think of the structure of your program independent from the syntax and grammar of particular languages.

What is a program?

If you followed along with the previous chapter, you have already written a program in the form of a shell script. There are many programming languages, each optimized for different trade-offs between ease of use, computational efficiency, portability, and specific tasks. The suite of popular languages has evolved over time, so that most of the widely used languages today were not even in existence twenty years ago. However, many key programming concepts and basic building blocks have remained largely the same.

Goals of the next few chapters

At first, writing a program may seem like taking a spelling test, a foreign language exam, a math test, and a car repair class—all at the same time. Learning the syntax of a programming language is the most conspicuous impediment to becoming a programmer, but it is actually more important to learn to think about your data analysis problems as a series of programmable tasks.

As an analogy, imagine you want to instruct someone in how to bake a pie. You might write instructions in English with U.S. units of measure—but you would be describing a set of underlying *tasks* that are themselves independent of language and measurement conventions. That is, the tasks required to bake the pie could be articulated in any language, regardless of syntax, vocabulary, or grammar. You

can think of the physical acts that go into cooking as similar to the building blocks of programming, and think of the words used to describe these acts as the terms of a programming language.

This book is not a cookbook—we do not provide you with lists of instructions for different tasks. This book is an introduction to how to make your own recipes. It might seem like a big step to go from not being able to cook at all to cooking from scratch without a recipe, but you can ease into it by starting simply, with only a few ingredients. You will grow familiar with programming by reusing the new skills you acquire and recycling and adapting bits of programs you write.

COMPILED VERSUS INTERPRETED PROGRAMS Programs are rarely written in a language that microprocessors can understand directly. Programs written in some languages, such as C and C++, are translated from human-writable and human-readable code (called **source code**) to computer-understandable instructions. Such translated programs are said to be **compiled**. Once the program has been compiled, it can be run again and again without being retranslated. The majority of programs that you use regularly are compiled in this way. The compiled program file is specific to a particular family of microprocessors and an operating system, and is not **portable** to another operating system. Also, other programmers can't modify the program or recompile it for a different type of computer unless they have the source code.

However, another category of programming languages exists in which programs typically are not typically compiled, but instead processed by an **interpreter** each time they are run. The interpreter is itself a compiled program that runs directly on the microprocessor. These languages are called interpreted languages, or **scripting languages**, and include Python, R, MATLAB, and Perl, among many others. There can be additional computational overhead to reinterpreting a program each time it is run, but even so interpreted programs have several advantages, especially for scientists. One such advantage is portability: the program file itself is the source code, so it is easier to modify, and can usually be run on any computer that has the correct interpreter. The shell scripts you wrote in previous chapters were interpreted programs, with the interpreter being `bash`. In the following chapters you will write programs for the `python` interpreter.

The distinctions between compiled and interpreted programming languages are discussed in greater detail in Chapter 21, along with instructions for compiling simple programs.

Practical programming

The scope of the next few chapters is modest compared to books dedicated to learning programming; we hope to convey a minimal set of skills that will get you started writing your own programs. These chapters are not intended as a comprehensive introduction to programming, or as a grounding in programming theory, but as a taste of practical programming. This will give you a footing for solving some of the simple problems you already face, as well as provide a departure point for getting started on more complex projects. The next step to taking on these projects would likely include more in-depth instruction, via either online resources or any of the many excellent books available for beginners. At the end of the next few chapters you should have a much better sense of which tasks are best approached by writing new software, what's entailed in writing a new program or repurposing an old one, and what specifically you would need to learn to take on more complex challenges.

The present chapter focuses on the basic building blocks of programs and introduces some commonly used terms (Table 7.1). Most of this discussion is not specific to any particular computer language, but we do lean somewhat towards Python, the language we will focus on in the following chapters. If you plan to learn programming with a language other than Python, it is still important to become familiar with the concepts described here. In Appendix 5, you can see examples of how these elements are applied in a variety of programming languages. Although this treatment is neither comprehensive nor general to all computer languages, it will form a foundation for translating data-analysis tasks ("I need to reformat each line of this file into tab-delimited decimal values") into programming terms ("Open file, read in each line with a `while` loop, parse variables, save formatted output"). After this overview, the subsequent chapters will explain how to use these building blocks specifically in the context of Python.

TABLE 7.1 Glossary of program-related terms

Term	Definfition
Arguments	Values that are sent to a program at the time it is run
Code	*Noun*: A program or line of a program, sometimes called source code *Verb*: The act of writing a program
Execute	To begin and carry out the operation of a program; synonymous with *run*
Function	A subprogram that can be called repeatedly to perform the same task within a program
Parameters	Values that are sent to a function when it is called
Parse	To extract particular data elements from a larger block of text
Return	In a function, the act of sending back a value; the value can be assigned to a variable by referring to the function name (e.g., in `y = cos(x)`, the function `cos` calculates and returns the value of the cosine of `x`, which is assigned to `y`)
Run	To execute the sequence of commands in a program; can also refer to the processing of a file by a program that finds instructions within it
Statement	A line of a program or script, which can assign a value, do a comparison, or perform other operations

Variables

The anatomy of a variable

Anyone who has taken algebra knows about variables. In essence, each variable is a name that holds a value. The value can be a number, a series of characters, or something even more complex. As the name suggests, variables can vary, by changing what pieces of information they hold or point to.

Each variable has several attributes. First, there is its **name**, usually a plain word that gives some indication of the variable content, such as `Sequence`, `Plot_Num`, or `Mass`. The programmer designates the name of the variable at the time the program is written. Some languages require a special symbol or symbols to be associated with the name of the variable; in Perl, for instance, variable names begin with `$`. Other languages, such as Python, forbid punctuation within variable names. Almost all languages make it illegal for the first character of a variable name to be a digit. Sometimes single letters alone are used as names, especially when the variable is keeping track of some internal value just for controlling program flow. In general, though, it is best to pick variable names that are distinctive and memorable, even if it means a bit more typing. This is one of the simplest steps you can take to improve the readability of your programs so that they can be understood later, whether by you or by someone else. To improve clarity, most variable names in this book will begin with a capital letter, so that they can be easily distinguished from other program components. (The exception is variable names only a single letter long; we have left these lowercase, since they can't be confused with anything else.)

The second variable attribute addressed here may initially seem a bit more abstract. This is its **type**, a designation of the kind of information the variable contains. Variables in modern computer languages each have a type, accommodating numbers, strings made up of text, images, lists of data, and so on. It is also possible to design and use entirely new types.

A third attribute for a variable is its **value**, that is, the actual piece of information that it holds. Not only can a variable be assigned a value upon creation, but it can be reassigned different values throughout the course of a program. A close relationship exists between a variable's type and its value, such that a variable of a certain type can only contain certain kinds of values.

Variables have other attributes that we won't discuss in detail here, but they can become important as programs get more complex. One of these is **scope**, which designates where in a program a variable can be accessed.

Basic variable types

There are a few variable types that you will use again and again; examples of these are shown in Table 7.2. You will notice that there are several variable types for numbers. This is because there are trade-offs between the size and precision of a number and the computational resources needed to store and manipulate the number. Having several number types may seem unnecessarily confusing, but it allows programmers to write code that is more computationally efficient and uses less memory; it also helps ensure that the program is behaving as expected.

TABLE 7.2 Commonly used variable types

Type	Example
Integer	`98`
Float	`98.6`
Boolean	`False`
String	`'Bargmannia elongata'`

Integer One of the most basic variable types is **integer**, which holds whole numbers without any fractional component (e.g., `0`, `-1`, and `255`). The limits on integer size are language-dependent, but in most languages integers can range from `-2,147,483,648` to `2,147,483,647`.[1] This range of possible values for an integer may seem so large that there is no chance that you could ever exceed it—and yet a timer that counts elapsed time in milliseconds would do so within a month. If you find yourself in a situation where the regular integer type is insufficient, you can go with one of the more specialized integer types: a variable of type **long** can hold much larger integers, but will take more memory, while a variable of type **unsigned** can go twice as high as a normal integer, but cannot hold negative integers.

Floating point Many scientific applications require numbers that either are extremely large, or else have decimal fractions. A useful type for these situations is **float**, shorthand for "floating point." This name signifies that the decimal point can float to any position, and not just lurk at the very end of the number. Like scientific notation, a float can represent a huge number or a tiny number using just a few digits (e.g., 4×10^{22}, the number of bacteria in a bioluminescent milky sea, or 1×10^{-15}, the diameter of a proton in meters). Floats are sometimes written as `4e22` or `4E22`. The limitations of floats mainly have to do with the number of significant digits they can accurately retain, rather than the size of the number. If a standard float isn't precise enough for your program, you can use a **double**, which occupies twice as much memory (hence the name) and has higher precision.

Boolean A variable of type **Boolean**[2] can have one of two values, `True` or `False`. Booleans are used for logical operations, such as the response to the question, "Is the value of variable `x` greater than variable `y`?" Booleans can also be interpreted as numbers. In this context, `True` is equivalent to `1` and `False` is equivalent to `0`.

Strings Most languages have a data type called **string** to handle a sequence of text characters. A string can include letters, digits, punctuation, white space (spaces, line endings, etc.), and other characters. Within a program, string values are almost always bounded by a pair of straight quotation marks, either single (`'`) or double (`"`). This is to differentiate text that is placed in a string from variable names and commands. Here are some examples of assigning strings to variables:

[1] Such limits are determined by the amount of memory used to store a variable; in the case of an integer with these limits, this amount is 32 bits. See Appendix 6 for more specifics on how numbers are stored and how this affects the way computers can work with them.

[2] The name "Boolean" is derived from the 19th century mathematician George Boole.

```
SequenceName = "Bolinopsis infundibulum"
Primer1 = 'ATGTCTCATTCAAAGCAGG'
DateString = "18-Dec-1865\t13:05"
Location = "Pt. Panic, Oahu, Hawai'i"
```

In each case, the text within the quotation marks on the right is placed in the variable named on the left. Some programming languages allow strings to hold nearly any character, while others cannot include certain characters. Unfortunately for international users, support for extended character sets (for example, ø, ü, °, ß, and even 力ナ) can require special kinds of string variables that support Unicode text encoding. Strings can be of arbitrary length—that is, they can contain no characters at all, a single character, or many characters.

Variables as containers for other variables

Arrays and lists

An **array** or matrix is a collection of data that lives under the roof of a single variable. Arrays can consist of one dimension, such as the set of integers from 0 to 9; two dimensions, such as a grid describing the pixels of a black-and-white photograph, or a record of depth, temperature, and salinity over time; or three or more dimensions. One-dimensional arrays are often referred to as **lists** or **vectors**. Depending on the computer language, an array or list can contain a mixture of types, including other lists. For example, a list called `morphology` could include floats, integers, string descriptions, and lists:

```
Morphology=[1, 0, -2, 5.27, 'blue', [4,2,4]]
```

Lists of lists can store multidimensional data, such as two-dimensional matrices with columns and rows. Lists come with useful tools for finding particular values, extracting elements, sorting data, performing some function on each datum, and other similar tasks.

Each item in an array can be accessed by referring to its location within the matrix, usually with an index number contained in brackets. In many languages, the first element is index `0`, rather than `1`. For a one-dimensional list such as `[A,G,T,C]`, the index is just the number referring the position in the list (in this case, `0` for `A`, `1` for `G`, `2` for `T`, and `3` for `C`). With a two-dimensional matrix that looks like a checkerboard (Figure 7.1), you can specify an entire row or column of data, or just a single element by giving its row and column indices. To change elements in a list, you can usually use the same syntax that is used to access the values, with the list item on the left and the value to be assigned on the right. For example:

$$A=\begin{bmatrix}2 & 7 & 6\\ 9 & 5 & 1\\ 4 & 3 & 8\end{bmatrix}$$

FIGURE 7.1 A multidimensional array

```
MyArray[2] = 5
```


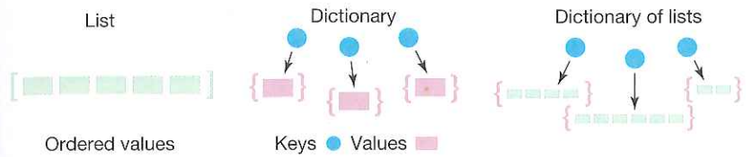

FIGURE 7.2 The structures of a list, a dictionary, and a dictionary of lists

Arrays are very important in scientific programming, making the code more efficient and easier to write and understand. If you find yourself repeating the same operation or calculation on a long series of values—especially if you are copying and pasting names like `Species1`, `Species2`, `Species3`—then this is the perfect time to learn how to store and process such variables as lists.

Dictionaries and associated arrays A **dictionary**, also known as an associative array, hash, or map, is another type of container for more than one variable. Unlike a list, which is simply a sequence of ordered values, a dictionary is a collection of names, or **keys**, with each key pointing to an associated **value** (Figure 7.2). The keys can be numbers, strings, or other types of variables. The keys for a dictionary must be unique, since each key can point to only one value, but different keys can have the same value. Here is some example Python code showing the creation of a dictionary, followed by the assigning of key-value pairs:

```
TreeDiam={} ← Create an empty dictionary with {}
TreeDiam['Kodiak'] = [68]
TreeDiam['Juneau'] = [85]
```

True to the dictionary analogy, values in dictionaries are looked up according to their keys, rather than by their position, as would happen in a list. For example, if you used a dictionary type to create and populate an actual dictionary, you could find the value for "cnidarian" using the word itself as the key, rather having to know that it was the 1024th item in the alphabetical list of definitions. In fact, the elements of data in a dictionary are in no particular order, and can be accessed *only* by looking up their key word.

Dictionaries are especially useful for gathering together data that describe some properties of a series of entities. For instance, you could create a dictionary containing all the molecular weights of amino acids, allowing you to easily retrieve the values for each amino acid by its name. Another handy trick is to combine dictionaries and lists, making a dictionary of lists. For instance, you could associate lists of DNA sequences from several different genes with particular specimen numbers.

Converting between types

Some programming languages, including C and C++, require the programmer to explicitly state variable types and then strictly enforce these types thereafter. In

other words, somewhere in the program before you use `x`, you have to specify that `x` is going to hold an integer. This makes for robust programs, since the computer never has to guess what type was intended, but it tends to reduce flexibility and require more code. Other languages, such as Python, set the type of the variable according to the values that are assigned to them, and only then strictly enforce these types. This is also quite robust, and means that the programmer still needs to think about variable types even though they aren't explicitly specified. Still other languages, for example Perl, try to take care of all type-related issues behind the scenes. The problem with this less restrictive approach is that code can break in very confusing ways when things go wrong. Although it can be convenient to have types handled entirely in the background, in many cases your intentions may be ambiguous and the computer's best guess is not what you expected.

This is particularly evident when you have a string containing a series of characters that could represent a number. For instance, take the string `"123"`. (Remember that the contents of a string are always contained in quotes.) To the computer, this is a sequence of characters, just like any other sequence, and not a number at all. A variable that contains this set of three text characters is very different from the integer 123; the latter is stored in the computer's memory as a binary representation of the number 123, not as a sequence of characters. If you were to add the integer 5 to the string `"123"`, for instance, would you expect to have the `"123"` treated as a number, resulting in 128, or the 5 treated as a string, resulting in `"1235"`?

In most programming languages, you would need to explicitly convert the variable types in this example to clarify the intended operation. That is, if you wanted the result to be the number 128, you would write the equivalent of, "Add `5` to a numerical interpretation of the string `'123'`"; whereas if you wanted the result to be the string `'1235'`, you would write the equivalent of, "Append a string interpretation of the integer `5` to the end of the string `'123'`." Because this kind of thing occurs often, there are commands built into nearly all languages to perform such conversions.

Variables in action

Programs need to do a lot more than just store information in variables. They also need to be able to do something useful with them. Often this consists of performing some kind of calculation to generate a new value. The tools for modifying or calculating new values include **operators** and **functions**.

Mathematical operators

Operators in computer programs include the mathematical operators that you are already familiar with: addition, subtraction, multiplication, division, power (x^y), and modulo (finding the remainder left after division).

The actions that operators perform depend on the language and the type of data they are operating on. Usually this is quite intuitive. Take the + operator:

If it is applied to two integers, it will return their sum. In many languages, + can also be applied to strings, as alluded to in the previous section, to produce a new string that contains the joined contents of the other strings. Although the operator is written as + in both cases, it is performing different tasks in each context.

The data that an operator acts on don't necessarily have to be of the same type. The + operator can add a float to an integer, for example, but not all combinations of data types are supported by all operators: the + operator cannot combine a string and an integer.

In some languages, including Python, operations on integers alone will often generate integers, perhaps even when you were expecting a float. The most trouble comes with the division operator, `/`. The precise result for `5/2` would of course be `2.5`, which needs to be stored as a floating point number. When the result is shoehorned into an integer variable, however, it truncates the decimal portion, leaving the somewhat confusing result of `2`, with the remainder having been discarded.

The operators discussed so far all do a calculation and return some new value as the result. One operator that we have taken for granted so far is the = operator, which assigns the value on the right side of = to the variable on the left side. This allows you to copy the value of one variable into another, or to store a new value.

Comparative and logical operators

Other operators make comparisons of variables and then return a Boolean value in the form of `True` or `False`. A common example is to test if one variable is greater than another. The results of operators can be assigned to a variable, or just used directly to help the program decide what to do.

Comparison operators can report whether two entities are the same or not. The equality operator is often written as `==`, and returns `True` if the entities on the left and right side of the operator have the same value. This is different from a single =, which assigns values rather than comparing them. Though it is common to think of = alone doing both of these tasks, most languages will not allow its use in both contexts.

In some languages, equality operators can even apply to strings. The statement `GenusName == 'Falco'`, for example, would return `True` if the variable named `GenusName` contains a string with the value `'Falco'`. These kinds of equality comparisons have to be exact, including whether the characters in the string are uppercase or lowercase. If you want to test whether a bit of text partially matches another string, there are other string-related functions which work more like the regular expressions of Chapter 2 and the `grep` command in Chapter 5.

Another useful operator which returns a Boolean value is the `in` operator (some operators are words, not punctuation marks). For example, the expression `x in A` returns `True` if the value of `x` is contained among the items in list `A`. If `A` is a list of several lists, then `x` must be a list as well, not one of the elements of those lists.

Other logic-related operators work with Boolean variables themselves. The names of these operators (`and`, `or`, `not`) describe their mode of operation pretty

well. These are important in testing several comparisons simultaneously, for example (`Depth > 700 and Oxygen < 0.1`).

Operators are evaluated according to a hierarchical order, per normal algebraic rules. Usually the order is exponents; multiplication and division; addition and subtraction; comparisons of equality; and finally, logical operators. Within these

TABLE 7.3 Common operators and their symbols

Operator*	Common symbols
Mathematical	
Addition	+
Subtraction	-
Multiplication	*
Division	/
Power	**
Modulo (remainder after division)	%
Truncated division (result without remainder)	//
Comparative	
Equal to	==
Not equal to	!=, <>, ~=
Greater than	>
Less than	<
Greater or equal	>=
Less or equal	<=
Logical	
And	and, &, &&
Or	or, |, ||
Not	not, !, ~

*Not all languages support all operator symbols, and in some cases the name of the operator is used instead of a symbol.

categories, operators are evaluated left to right. This order can be modified with parentheses, grouping operations that should be performed first. The most commonly used operators are listed in Table 7.3.

Functions

Functions can be thought of as little stand-alone programs that are called from within your own program. Many functions for common tasks come built into programming languages. You can also write your own custom functions; these can be reused as often as you want, both within a program and potentially later on by other programs that you write. Functions can be defined locally in your program, or they can be stored in external files and loaded into your program as needed. Functions will be an important part of all but the simplest of the programs you write.

Functions can accept variables, which are then referred to as **parameters** of the function. These are usually sent to the function within parentheses, `()`. Functions can also return values. For example, the `round()` function, as used in the context of `y = round(2.718)`, takes a number with a fractional portion (in this case the floating point value `2.718`) and returns the value after the decimal is rounded off (here, rounded number is assigned to the variable `y`). Even when functions aren't designed to take any parameters, a set of trailing empty parentheses, e.g., `PrintHelpInfo()`, is still usually needed to differentiate the name as pointing to a function, rather than to a variable or other code element.

Flow control

Decisions with the `if` statement

At this point you know a bit about variables and how to obtain or create values based on their state, through comparisons and calculations. With just these tools alone, though, you couldn't write much more than a glorified calculator. The real power of programming becomes apparent with conditional decision-making—the ability for a program to follow different courses of action depending on the state of variables.

The `if` statement is the most widely used building block for such decision-making. It is like a sign at a fork in the road. If the statement on the sign is true, then you take one fork, and if it is false you take another route, eventually rejoining the main sequence of commands within the program. In programming terms, the `if` statement evaluates an expression that returns a Boolean. If that expression is true, it then executes a specified set of commands. You can use an `else` statement in conjunction with an `if` statement to execute another set of commands when the expression is false. You can also chain together a series of `if-else` statements to combine several of the logical conditions into a complex sequence of events. Figure 7.3 depicts both an `if` and an `if-else` statement.

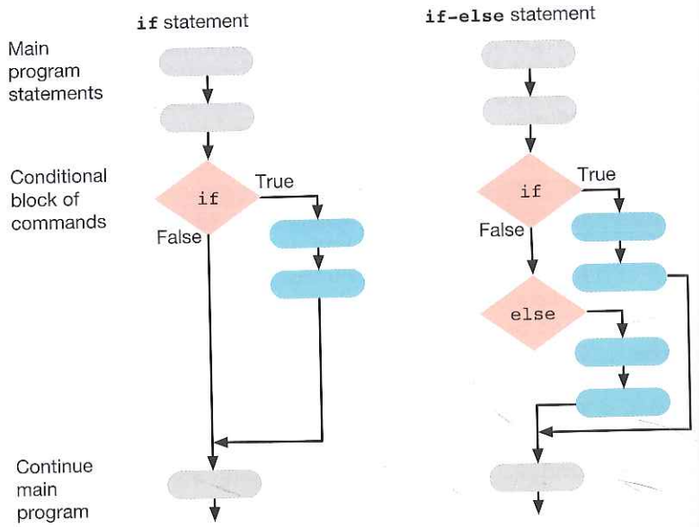

FIGURE 7.3 The flow of a program through conditional statements The ovals represent program statements and the diamonds represent decision points. Arrows represent the operation of the program under different circumstances. In an `if-else` statement, for example, if a statement is evaluated as `True`, you take the `if` fork; but if it evaluated as `False`, you take the `else` fork.

Below is an example of an `if-else` statement as it would be written in Python; other languages will have slightly different ways of stating the same thing:

```
A = 5
if A < 0:
    print "Negative number"
else:
    print "Zero or positive number"
```

Looping with `for` *and* `while`

The building blocks presented so far allow you to write a program consisting of a linear sequence of events to be executed from top to bottom, with the power to control which events in this sequence are executed. A linear program like this can automate a complicated sequence of analyses, but it isn't well-suited to repetitious tasks that require the same computations to be made over and over. This, however, is the exact kind of task you usually want to write a program for. The power of a program to plow through hundreds of calculations in only a few lines of code is realized by **looping** repeatedly through the same basic set of commands. A `for` loop and a `while` loop are illustrated schematically in Figure 7.4.

The `for` loop **Loops** are portions of programs that are repeatedly executed until some condition is met. Of these, by far the most widely used is the `for` loop. A `for` loop cycles through each item of a predefined collection, performing a series of commands each time it does so. The items it cycles through can be drawn

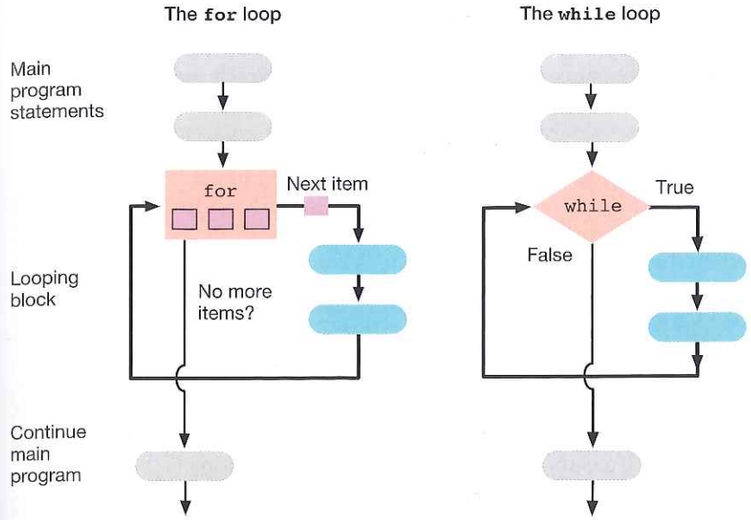

FIGURE 7.4 The flow of a program with a `for` or `while` loop The orange box represents a collection of objects which are taken one-by-one and operated on by the blue ovals.

from a variety of collections, such as a range of numbers from 1 to 100, a list of species names, an archive of protein sequences, the lines of a data file, and so on.

The power of a `for` loop comes from being able to repeat the same operation on each item in a long list. In fact, some programming languages have a specific `foreach` statement emphasizing this item-by-item capability. Loops can also be nested, or placed one inside the other. For instance, you might use three nested `for` loops to step through each amino acid of each protein sequence in each file of a folder.

As much as any programming element, the syntax used in `for` loops varies from one language to another. Many examples of this variation can be seen in Appendix 5. Two examples in particular are shown in Figure 7.5, one from Python and the other from C; note how verbose the C code is compared to the Python code. This may help you understand why we prefer Python for our work, and why we chose it as the programming language for this book.

(A) `for` loop in Python

```
for Num in range(10):
    print Num * 10
```

(B) `for` loop in C

```
for (Num=0; Num < 10; Num++) {
    printf("%d",Num * 10);
}
```

FIGURE 7.5 Code snippets showing a `for` loop in Python and in C

Innumerable situations can be tamed with `for` loops: processing every file that meets a certain criterion, splitting each line of a data file into its component parts, translating each gene sequence in a FASTA file, or performing a unit conversion on each entry of a column of numbers in a data-logger file.

The `while` loop In some situations, you enter a loop without having a predetermined list to cycle through. When a loop is open-ended like this, you can use a `while` loop to continue cycling through until some logical condition is evaluated as `False`. Like an `if` statement, the `while` loop begins with a logical test, but unlike the `if`, when the conditional block of code is complete, the program does not return to the main path, but loops back up and checks the condition again.

At least one variable involved in the test expression must be modified by the statements in the loop, or else the loop will keep cycling forever. This is the infamous "infinite loop." To avoid this situation, programmers sometimes link in a second fail-safe condition. Typically this is a counter—that is, a variable which keeps count of how many times the loop has been traversed. The loop can be made to exit when the count exceeds the expected maximum iterations.

Situations where `while` loops are useful include improving an approximation until it reaches a sufficient level of precision, waiting until a temperature or environmental condition is reached, continuously monitoring a sensor until a stop command is given, and reading lines from a file until the end is reached.

Using lists and dictionaries

Lists

Think of a list as a series of numbered boxes that start with box 0. You can create a list with a given number of boxes, and later append more boxes to the end of this series if you need more containers. A list that includes a box numbered 16 will necessarily have a box numbered 10. If you try to access box number 94 when there are only 70 boxes, though, an error will be generated. The data in a list can be accessed one at a time, or ranges of data can be accessed at once. You can peek inside any box you like, replace the contents of a box, remove or insert boxes into the beginning or end of the list (shifting the remaining boxes), or even reorder the boxes. After such modifications, the number used to access each box depends only on its position.

When do you use lists as opposed to singular variables? Pretty much any time you are handling more than one piece of closely related information. Imagine, for instance, a field site at which you measure the diameter of 5 trees. You might first think to load these values into a series of 5 variables named `Diameter1`, `Diameter2`, `...`, `Diameter5`, make calculations on each (`crosssection1=`, `crosssection2=`), and then print each result with five separate `print` statements. This is a cumbersome solution, since you would need to repeat a nearly identical set of commands to act on each variable independently for each tree. It is cleaner and easier to create a list variable called `Diameter` which holds the five values in a list. By using a `for` loop, you can then easily write one set of commands which performs the same operations on each measurement in turn.

Lists also have the advantage of making your code more general. In our example, you may later want to use the same program to analyze a dataset with 6 measurements per site. If you used a different variable name for each measurement, you would have to add a variable to your program to accommodate the new dataset. If you use a list, however, your program would require no additional modifications. A list can handle thousands of values just as easily as it can handle ten.

Dictionaries

Dictionaries may seem a bit less intuitive than lists at first, but it is likely that you will find them to be equally useful, if not more so. Recall that the data entries of a dictionary are looked up with unique labels known as keys. If you wanted to perform an operation using each entry in a dictionary, you would first get a list of the keys, and then use those to retrieve each dictionary item one by one. Since in a dictionary, data elements don't have a fixed position, sorting the entries within the dictionary is not meaningful the way it would be with a list. However, sorting the keys and then retrieving the associated values in a particular order is often useful.

The unordered nature of dictionaries makes them more flexible than lists in several important ways. In a dictionary, you can create an entry for key `16` without having any data corresponding to keys `0` to `15`. This lets you fill in data as they become known. Although list indices must always be integers, dictionaries allow you to associate values with keys of many different variable types. For instance, you could use a dictionary where the keys are strings, each string being a specimen name:

```
TreeStat = {} ← Create an empty dictionary to build upon
TreeStat['Kodiak'] = [ 68, 57.8, -152.5]
TreeStat['Juneau'] = [ 85, 58.3, -134.5]
TreeStat['Barrow'] = [133, 71.3, -156.6]
```

Dictionaries and lists can be combined in useful ways. Imagine that each field site where you are measuring tree diameters has a unique name. You could create a dictionary `TreeStat` that uses the name of each site as the key, with a whole list of measurements and coordinates as the associated value for each entry. Such dictionaries of lists provide an easy way to store and return a large set of related measurements.

Other data types

One of the other basic data types, the **string**, is actually quite like a list in many respects. It is, after all, an ordered sequence of characters. You can step through each character of a string with a `for` loop, and you can even extract substrings or certain elements from a string in the same way you would extract subportions of a list. A difference is that you can't always modify individual elements of a string

the way you can those belonging to a list. Fortunately, there are other ways to carry out these kinds of operations on strings, as you will learn in later chapters.

Finally, there are other data types that hold collections of objects, such as sets and tuples in Python; however, these will not be addressed here.

Input and output

User interaction

Most programs provide for some means of user interaction. The vast majority of computer users have interacted with programs only through graphical user interfaces. Programs that are run at the command line (including the programs that you will soon write) also provide for several means of user interaction. Program options specified at the command line when a program is launched are called **arguments**. The command `ls`, for instance, can be run with several arguments that control what files are displayed, and in what format. These arguments are specified by listing them after the command when it is executed, for example `ls -a` or `ls *.txt`.

Arguments are a mode of user input for programs that first gather all the information they need, and then go about their business in one big push. This is good for the user because it isn't necessary to babysit the program while it runs, or to respond to queries during the execution. This is especially important for programs that take a long time to finish. If your program is configured with arguments, then it is also easier for other programs to control it. This will be important if you want to build up workflows that automatically shuttle data through a series of programs. Python and many other languages have built-in tools for passing arguments from the command line to variables within the programs you write.

Some command-line programs are designed to respond to user input while they are running. Command-line text editors, for example `nano`, are an elaborate example of this. Simple interactive interfaces can be desirable in some cases, and like arguments, they are simple to implement in your own software. For instance, you will create an interactive interface for the DNA analysis program described in the next chapters; this will allow you to calculate basic molecular properties of short sequences that you enter at the interactive prompt.

User interaction isn't just about getting input from the user; it is also about generating output, as well as giving the user feedback about the progress and results of the program. Typically, the command used to send output to the screen is either called `print` or some variation of that word. Output can also go directly into a file, a process we discuss next.

Files

Like variables, files associate some chunk of data with a name. Unlike variables, which have a fleeting existence while the program is running and thus live only in the computer's volatile memory (RAM), files reside on the disk drive. They are still there after the program stops, and even after the computer is turned off. While

variables are only accessible from within the program that created them, files are available to any program with access to the disk and permission to read them. Files therefore serve both as a long-term repository of data and a way to shuttle data between programs. Reading and writing files is an essential part of the programming you will do for your scientific research.

There are two separate levels of operations that must be completed to access data in a file. The first level consists of all of the mechanical details of gaining access to the contents of the file from within a program, such as finding it on the disk and loading its contents into memory. Fortunately, programming languages provide tools and commands that take care of all this work behind the scenes. In order to gain access to the contents of a file, you usually need to indicate only the path to the file and whether you want to read from it or write to it.

The second level of operations required to access data from a file is to establish relationships between the data in the file and variables in the program. This works in both directions. That is, when reading a file, you are **parsing** its data from a file into program variables, whereas when writing to a file, you are packaging data from program variables.

For example, a data file might contain lines of text like the following:

```
ConceptName     Depth  Latitude  Longitude Oxygen TempC  RecordedDate
Thalassocalyce   348.7  36.7180  -122.0574  1.48  7.165  1992-03-02 17:40:09
Thalassocalyce   520.3  36.7491  -122.0368  0.52  5.826  1992-05-05 20:01:10
Thalassocalyce   118.4  36.8385  -121.9676  1.52  7.465  1999-05-14 17:48:27
Thalassocalyce   100.9  36.7270  -122.0488  2.30  9.497  1999-08-09 20:12:11
Thalassocalyce  1509.6  36.5846  -122.5211  0.95  2.774  2000-04-17 20:04:23
```

This information, stored on the disk, will be most usable in the programming environment if each column can be loaded into list variables of appropriate types (strings, integers, floats, etc.). Conversely, once a calculation or conversion is performed, the variables will likely be written out to a file and saved.

Parsing and packaging file data usually comes down to string manipulation. The text is read from a file as strings, usually line-by-line, and the data are then extracted from these strings and placed in the appropriate variables. Regular expressions, the first tools you learned in this book, are often used for this process. Writing data to text files is the opposite process: data from variables are converted to strings, joined together with the appropriate formatting characters, and then written to the text file, once again line-by-line. The specifics of how you parse and package data will depend on the format of the data within the file, meaning that data from different sources, databases, or instruments will often require that you write or modify a small program to handle conversions and calculations. This, then, is where the heavy lifting of file handling occurs when you are programming.

In addition to storing data for analysis, files can also be a means of controlling program behavior. You have already seen how a text file can store a list of program commands as a script. Many programs can take their raw input and march-

ing orders from a file, after which they send their output directly to another file. Text files can also serve as a record of what a program did; in such cases, they are called log files, or just plain logs.

Libraries and modules

Computer languages have many built-in tools available out of the box. These include some of the basic building blocks we have mentioned—predefined variable types, basic functions, and simple mathematical and logical operators. For more specialized tasks, additional functions can be written and then bundled together in **modules**. By **importing** from one of these modules, you can gain access to the suite of functions stored within. Some of the most commonly used modules provide tools for more advanced mathematical functions (e.g., sin and log), interacting with the operating system (e.g., listing directory contents and executing shell commands), retrieving content from the Internet, performing regular expression searches, working with molecular sequences, and generating graphics. Python modules are explored in Chapter 12.

Comment statements

Programming languages provide for comments—that is, portions of text that are helpful to human beings but are to be ignored by the computer when the program is run. Comments are usually marked by starting the line with a special character, such as #, //, or %.

Comments serve several important functions for programmers. Commented blocks of text are used to provide documentation and a description of both what a program does and how to run it. Commented text at the top of a script usually includes a list of any arguments the program may take, and a revision history of changes, so that updated versions of the script can be distinguished.

Within the program, comments may be associated with individual lines or with subsections of code, to explain how they operate. These internal notes are important for future readers of your program (including yourself), who typically are trying to determine (or maybe just remember) how the program operates.

Comments are also useful for troubleshooting your program. You can isolate problems by turning portions of the code on and off with comments, to see if an error still occurs in the same way.

Objects

A full treatment of **objects** is beyond the scope of this book. However, objects are essential components of contemporary programming, and aspects of these components are woven into the fabric of many programming languages; thus, even a simple treatment needs to mention them.

An object is a sort of "super variable" that can contain several other variables within it as part of its definition. In fact, not only can objects contain variables in the traditional sense, but functions as well; in this context, these are called **methods**.

Consider a bicycle as an object. Your own personal bicycle may be just one example or instance of the platonic ideal of a bicycle. Likewise, your bike has many properties associated with it. In computer terms, you access the nested properties of an object using a **dot notation**, where the name of the property is joined to the name of the object with a period:

```
MyBike.color ← Color of your bike
MyBike.tires ← Properties of your tires
```

Used in this way, these statements might report the status of those properties, returning `'blue'` for color, as well as various values for the brand, air pressure, diameter and tread of your tires. These are examples of **nested** properties:

```
MyBike.tires.pressure ← A nested property of your tires
```

Methods (that is, functions) can also be associated with the object; if so, these are accessed with dot notation as well. These methods can be little programs, which take in values and apply them to the bicycle object:

```
MyBike.steer(-4) ← Steer 4° to the left
MyBike.color('red') ← You can often set and read with the same notation
MyBike.pedals.pedal(100) ← With the pedals, pedal at a speed of 100
```

The reason that these object-related ideas are important is that in many languages, even simple variables are objects and have their own methods and properties. Sometimes instead of using a separate function and sending your variable to it as a parameter, you can access a method from within the variable itself.

For example, imagine you have a string variable and you want to convert it to uppercase letters and print it. You might find a function `uppercaser()` and send your string into it:

```
MyString='abc'
print uppercaser(MyString)
```

Alternatively, the string variable itself might contain an uppercase function which you can access:

```
print MyString.upper()
```

There are many more implications and unique properties of object-oriented programming that are beyond the scope of this book. The main significance of objects for the topics we will address is that some functions you will use are methods of variables rather than stand-alone functions.

SUMMARY

You have learned:

- The differences between compiled and interpreted programs
- That variables have several attributes, including name, type, and value
- Some of the variable types widely used among programming languages
- That groups of variables can be stored in ordered lists and unordered dictionaries
- The basics of using variables with operators
- Key elements of the structure of programs, such as loops and functions

Chapter 8

BEGINNING PYTHON PROGRAMMING

Now that you have a passing familiarity with some of the basic components of programming, it is time to put them into practice. This will require applying some general programming concepts in the context of a particular language: Python. Within the next few pages you will learn how to make and use basic Python programs.

Why Python

In this book, you are going to write your programs in the popular scripting language Python. There are many programming languages available to scientists, and a brief explanation of our choice of Python will help you understand a bit about the language and the goals of these next chapters. Python is a relatively new language, but it has rapidly grown in use. It is now widely taught in introductory programming courses, and many students and researchers will likely encounter it in this and other contexts, reinforcing the information we present here. Many of the programs written and distributed by scientists are in Python, which makes Python directly relevant to the real-world computing environment in biology. Python is also widely used in industry, with considerable intellectual and financial support for further development coming from Google and others.

Beyond a large and growing user base, the reasons most often cited for learning and using Python are the relative clarity of the code, the ease of manipulating text data, and its native support for advanced features such as object-oriented programming. Newcomers tend to get up and running quickly in Python, yet it is also a fully functional and mature language suitable for complex tasks.

In key respects, Python is most directly comparable to Perl, another language that is also well-suited to text manipulation and which has been widely used by biologists. Emotions can run high when discussing the relative merits of the two languages. Python does have some shortcomings, and we ourselves have used Perl in our past research. Despite this, we have selected Python for this book for

two primary reasons: First, Perl code can be difficult to read and understand, especially for newcomers; this is due in part to its heavy reliance on punctuation characters with special meanings. Second, the growing momentum behind Python provides more opportunities to take advantage of other computational and scientific resources, such as existing code and documentation, and integration with ongoing projects and course work.

Among other essential tasks, Python programs can reformat and organize data files, perform mathematical and statistical analyses, and assist with visualizing data and results. This book, however, will emphasize data handling over analysis and visualization. As discussed in *Before you begin*, most of the day-to-day computational challenges encountered by biologists consist of reformatting and reorganizing data files. This is a natural consequence of the explosive growth in dataset size across the field, which precludes manual manipulations that might have worked in the past. Not only are these manual operations tedious and time consuming, but they can't legitimately address the sophistication of interdisciplinary analyses, which require modifying data files to enable cooperation between programs that weren't designed to work with each other.

Even if you are already familiar with Matlab, R, or some other language, Python's simple power will complement those tools, as you wrangle your data into an optimal format for further analysis. So while you could build on the data manipulation skills we present here to write sophisticated new Python analysis and visualization tools, why reinvent those technologies? A better use of your new skills will be to prepare your datasets for further analysis, performed by packages dedicated to those tasks.[1]

Writing a program

If you have skipped through the last chapters, this is a good place to rejoin the narrative, but you first need to make sure that your programming environment is set up correctly. If you haven't already done so, create a `scripts` directory and edit your `PATH` variable as described in Chapter 6.

In the coming pages, it is helpful to follow along by testing the examples as you read. You can type these in as you go; the final programs and data files are also available in the examples package, which you can download. The added value to typing them in is that because you will have built up a set of programs that you understand intimately, you will be more easily able to modify them later for your own purposes.

Appendix 1 describes how to install and use Python on Windows.

Getting a program to run

If you are running Mac OS X, Linux, or most any other flavor of Unix, your system is already set up to run Python programs. You will be able to create a blank text file

[1] If you *really* end up liking Python and want to use it for analyses also, you might look at `matplotlib`, briefly introduced in Chapter 12, which provides MATLAB-type graphing commands to the Python environment.

and quickly set it up to operate as a Python script. If for some reason you have to install a new copy of Python on your system, use Python version 2.6, even for Mac OS X. Remember also that as you work through examples, a `blue background` will indicate shell commands entered in a terminal window, and a `tan background` will indicate scripts edited in a text editor.

From Chapter 6 you are already familiar with creating scripts—that is, text files containing a series of commands to be executed by your computer. Instead of being interpreted as a series of `bash` shell commands, as your shell scripts were, these next scripts will be read by the `python` program. Like a `bash` script, the first line of the file needs to tell the system where to send the rest of the file contents. In this case, you will begin the file with the shebang line (`#!`) followed by the location of the `python` program.[2] You can find the location of the Python program with the command `which python`:

```
host:~ lucy$ which python
/usr/bin/python
```

This gives the absolute path to `python`, and it would work fine on your computer. However, hardcoding absolute values for paths and other variables into your scripts, as this practice is called, can be a problem if you share those scripts with colleagues. For example, they may have `python` installed somewhere else on their computer. You will therefore want to use a more flexible method to indicate where the `python` program is located.

Nearly all systems have another program called `env` (similar to `which`) that can find `python` or any other program for you. Instead of sending your scripts to `python` directly you will send them to `env`, and ask it to pass them along to `python`. To do this, the first line of the file will read:

```
#! /usr/bin/env python
```

Notice that there is no slash after `env` (because it is a program, not a folder) and that there is a space between `env` and `python` (because you are telling `env` to find and run `python`, wherever it might be).

To begin a new Python script, open your text editor, create a new empty document, and type the shebang line above into this document.

Constructing the `dnacalc.py` program

As the first exercise you will build up a program which takes a string of text representing a DNA sequence, made up of the bases A, G, C, and T, and prints out information about the sequence, including the percent composition within the se-

[2] If this doesn't make sense to you, go back and read Chapter 4 on the shell and Chapter 6 on scripting. You will need to have your system set-up properly and be familiar with the basic skills which we described in the earlier shell chapters to function effectively here.

quence of each base. To do this you will need to determine the length of the sequence, count the number of each of the bases, and do some simple arithmetic. To present the results, you will print the numbers to the screen.

To get you up and running as fast as possible, this example draws on portions of many different aspects of Python. If you get stuck at any point, you can consult with other readers of the book at `practicalcomputing.org`. You can also refer to Chapter 13, which covers general debugging practices and approaches, as well as specific `python` error messages and their likely causes.

Simple `print` *statements*

In your script, type the short program that follows. Be sure to include the shebang line, and to make sure the capitalization of each command is right. You can add blank lines between sections of the script, but be sure not to add any extra spaces at the beginning of the lines:

```
#!/usr/bin/env python

DNASeq = 'ATGAAC'
print 'Sequence:', DNASeq
```

Now save this file in your ~/scripts folder with the name dnacalc.py. The colors of the text should change when you save it, indicating that the editor (for example, TextWrangler) has recognized it as a script. Because of the modifications to the `$PATH` that you completed in Chapter 6 on shell scripting, the scripts folder has special properties, so be sure to save the file here.

Open a terminal window, `cd` to your `~/scripts` directory and then use `chmod` to make the file executable:[3]

```
host:~ lucy$ cd ~/scripts
host:scripts lucy$ chmod u+x dnacalc.py
```

Remember that you can use the tab key to auto-complete a file name after you have typed the first few characters. Also, if you are not sure how a particular shell command works, you can look up the manual page using the `man` command, for example `man chmod`.

You now have a file named dnacalc.py which is ready to execute. Type its name, press return, and see what happens:

[3] If for some reason you find yourself working and storing your script within a folder that is not part of your path, you will have to run the program using the command `./dnacalc.py` or else it won't be found. The dot indicates the current directory (similar to how `..` means the enclosing directory). Even though you may be working in the folder which contains your program, the shell will not know where to find the command you are typing if it is not in your path.


```
host:scripts lucy$ dnacalc.py
Sequence: ATGAAC
host:scripts lucy$
```

Did it print out the text `Sequence: ATGAAC`? If so, congratulations. If not, then an error message may have been generated that will help you pinpoint the problem. If there is an error, does it indicate that the program was not found, or does it perhaps say "permission denied"? If so, there is probably something wrong with the way your path is configured, where the program is saved, or the permissions that were set with `chmod`. If the error says something like `Traceback:`, then that is actually a good sign. It means that it tried to run your program, but there was an error in the code itself. Check for typos, including extra spaces. If you get `IndentationError: unexpected indent`, then check that each line has no extra spaces at the start.

NAMING VARIABLES Even for experienced programmers, it can be difficult to read through a new program and figure out what words are variable names created by the programmer and which ones are built-in parts of the programming language. Text editors can help with this by automatically displaying variable names as a particular color. The best way to avoid this problem, though, is to choose your names well, or use a convention which distinguishes them from the style of the reserved words. Because nearly all Python functions start with a lowercase letter, you can capitalize the first letter of your variable names, as we have done here.

This program first creates a variable named `DNASeq` and puts the string `'ATGAAC'` in it. The print statement then displays three different things all on one line—the literal string `'Sequence:'` (this text is used directly and never placed in a variable), then the contents of the variable `DNASeq`, and then an end-of-line character. You didn't have to tell Python to add the end-of-line character, for it does that on its own (though this behavior can be turned off). A comma separates the two pieces of text that are given to the `print` command for display. Later, we will explore other ways of formatting and joining pieces of text that provide a bit more control. Also, just as the line ending is added automatically, a space is added between the pieces of information that are separated by a comma.

Python interprets any text within straight quote marks, either single (') or double ("), as a string. This is to differentiate the contents of the string from the computer code itself. The advantage of having both types of quotes at your disposal is that it is easy to include a quote mark within your string without inadvertently terminating it. For example, this works:

```
MyLocation = "Hawai'ian archipelago"
```

but this would not work:

```
MyLocation = 'Hawai'ian archipelago'
```

The second single quote, between `i` and `i`, is interpreted as the close of the first one. The characters following the second quote are thus considered to be outside of the string. Most text editors designed for programming will help out by showing open strings in a conspicuous color.

FURTHER TROUBLESHOOTING If the `print` operation seems to be the source of your problem, check what version of Python you have installed. You can do this in a terminal window by typing `python -V` (make sure to use a capital V). The scripts presented here are designed to work with versions 2.3 through 2.7; there are no versions 2.8 or 2.9. Although Python 3.0 has been released, it contains some fundamental changes in the language and has not yet been widely adopted at the time of this writing. If your version is 3.0 or greater, try putting parentheses around the material that follows the print statements. If you reach an impasse, visit the forum at `practicalcomputing.org`.

You did not have to explicitly state that the `DNASeq` variable is a string when you created it. Python figured that out when you assigned a string to the variable. Likewise, `print` recognized `'Sequence:'` as a literal string rather than a variable or function name. In both cases, this is because the text is within quotes.

The `len()` function

Now that you have a variable holding a DNA sequence, you will do some calculations to summarize some attributes of the sequence. These calculations will take advantage of built-in functions. Functions are miniature programs or commands that are available within your program. A function can take input from variables sent to it as parameters (usually contained within the parentheses that follow the function name), perform some task (often a calculation involving the parameters), and return one or more results. The names of functions are case-sensitive, so it is important to use the correct capitalization. A space between the function name and the parentheses is optional in Python, but is not typically used.

The built-in `len()` function takes as a parameter an object such as a string, list, or dictionary; it then returns the number of elements in the object, which by convention is considered to be the object's length. In this case, it will return the number of characters in a string you give it. Back in the editor window, add the following two lines to the end of the program, save it, and then run it again (using the ↑ in the terminal window to recall your previous command):

```
SeqLength = len(DNASeq)
print 'Sequence Length:', SeqLength
```

When you re-execute, you should get two lines of output: the first line again showing the sequence, and the second line indicating its length. If not, make sure that you saved the changes you made to the program before running it.

In your program, the variable `DNASeq` already stores a string, so it is printed as-is. In the second case, the `SeqLength` variable holds an integer, and the Python `print` command translates it into a string for display. This automatic translation by `print` works with numbers, letters, and even calculations, converting them to strings on the fly. However, it only works when a single data type is involved, or when different types are separated by commas. Other formatting strategies require the explicit conversion of non-string data to strings, as you will learn next.

Converting between variable types with `str()`, `int()`, and `float()`

Python has the ability to do **mathematical operations** on numbers using typical operators for addition (+), subtraction (-), multiplication (*), and division (/). You can also do exponents (**) and many other operations.

The command `print 7+6` returns `13`, and `print 7+3*2` also returns `13`. Just as with algebra, the sequence of operations in your formula gives precedence to exponents, then multiplication and division, then addition and subtraction. Also as in algebra, you can control the order of these operations using parentheses, so `print (7+3)*2` returns `20`. Parentheses in expressions such as `7+(3*2)` can often help clarify what a long formula or logical expression is actually doing, even though they are not necessary.

Python also allows addition and multiplication with strings. In this context, `print '7'+'6'` returns `'76'`, and `'a'+'b'` returns `'ab'`. Note that there are quotes around the 7 and 6; the numbers in this case are not naked, as they were in the previous examples. The quotes indicate that Python should interpret the numbers not as integers (actual numbers) but as strings (bits of text), in this case single characters. Where adding integers gives their sum, adding strings returns the joined strings.

The + operator is a useful and easy way to join together multiple strings into one. However, it is not legal (that is, not permitted by Python) to add together a mixture of strings and numbers, because the correct interpretation is unclear. The statement `print '7'+3` will fail because it is impossible for the python interpreter to know whether you want the output to be the integer `10` or the string `'73'`. If you want to use + to build up a string from variables of different types, you must first convert each variable to a string, and then join the strings together. The `str()` function helps with this by converting other data types, including numbers, to strings. In a program, the statement:

```
print '7'+ str(3*2)
```

would display the string `'76'`.

Here, 3 and 2 are both integers, so they are multiplied by `*`. Their product, `6`, is then converted to a string by `str()` and joined to `'7'` to produce a new string, `'76'`.

Just as it is often necessary to convert numbers to strings when formatting output, it is often necessary to convert strings to numbers when you gather input from a user or a text file. The `float()` function converts a string or integer to a floating point number (defined in Chapter 7). For example, this statement in a script:

```
print float('7.5')+3*2
```

would return `13.5`.

The `float()` function is flexible, in that it can interpret scientific notation as well. So this statement:

```
print float('2.454e-2')
```

would return `0.02454`.

The `int()` function converts a string or float to an integer. If the value being converted contains any decimal fraction it is truncated, not rounded. Most mathematical operations that combine floats and integers, including addition, subtraction, multiplication, and division, return a float.

The built-in string function `.count()`

As described in Chapter 7, some variable types have functions built into the variables themselves. These built-in functions, known as methods, are accessed using dot notation. Strings have a method called `.count()`, which counts how many times a particular substring occurs in a string. The statement `print DNASeq.count('A')`, if inserted into your program would output the value 3. Modify the program to add the following lines:

```
NumberA = DNASeq.count('A')
NumberC = DNASeq.count('C')
NumberG = DNASeq.count('G')
NumberT = DNASeq.count('T')
```


Using four different variables that each need to be independently analyzed is not a very efficient way to do this operation, but for the moment it is sufficient. In later chapters, you will learn how to store similar data like these in a list, and then write your analysis process once and use it on each element of the list.

With these commands, you create four new integer variables. Each contains the count of how many times the corresponding nucleotide is found in your sequence string. You could provide the user with these raw counts, but they are typically more informative when displayed as fractions. To determine fractions, you will next divide each count by the total number of nucleotides in the sequence, which you already determined and stored in the `SeqLength` variable above.

Math operations on integers and floating point numbers

Remember from Chapter 7 that numbers with decimal components are called floats. This brings us to one of the foibles of the Python language: in Python 2.7 or earlier, an integer divided by another integer is truncated to give an integer result. So in Python, `5/2` gives the answer `2`. A mixture of integers and floating point will return a floating point number, so `5/2.0` gives the correct answer of `2.5`. We admit that this is potentially troublesome behavior, and it has been modified in Python 3.0. For the moment, the solution is to just make sure that there is at least one floating-point element in your statement so that the answer is provided as a float as well.

In addition to converting strings to floats, `float()` can also convert integers to floats. Modify the `SeqLength` line of your code to include the `float` function as follows:

```
SeqLength = float(len(DNASeq))
```

Functions can be nested—that is, placed inside one another. In this case, the result from the function `len()` is evaluated first and used as the input to the function `float()`. The output of *that* function is what is assigned to the variable `SeqLength`. This is a bit like the pipe at the command line: data can be processed through several steps without writing them to variables or files at each stage. With this modification, the variable `SeqLength` is now a float rather than an integer. If you run the program at this point, notice that the length is now reported as `6.0` instead of `6`, indicating that it can have a fractional part (even though that fractional part is currently `0`).

Since `SeqLength` is used in each calculation, changing it to a float is sufficient to ensure that all of the subsequent division operations will also result in floats. To avoid having to use `str()` to convert the resulting fraction to a string, we'll combine the print operation with the calculation into one line. At the end of the existing script, add in a print statement like this for each of the four nucleotides:

```
print 'A:', NumberA/SeqLength
```

Here is the whole program up to this point:

```
#!/usr/bin/env python

DNASeq = 'ATGAAC'
print 'Sequence:', DNASeq

SeqLength = float(len(DNASeq))

print 'Sequence Length:', SeqLength

NumberA = DNASeq.count('A')
NumberC = DNASeq.count('C')
NumberG = DNASeq.count('G')
NumberT = DNASeq.count('T')

print 'A:', NumberA/SeqLength
print 'C:', NumberC/SeqLength
print 'G:', NumberG/SeqLength
print 'T:', NumberT/SeqLength
```

Run the program at the command line and verify the output:

```
host:scripts lucy$ dnacalc.py
Sequence: ATGAAC
Sequence Length: 6.0
A: 0.5
C: 0.166666666667
G: 0.166666666667
T: 0.166666666667
host:scripts lucy$
```

You might also verify for yourself what happens if you try the division using an integer value of `SeqLength`. Copy the line where it is defined and paste it below. On this copy, remove the float function from the equation, and run it again. Because the more recent assignment overrides the first one, you should see a different output, with all zeroes for the percentages. In each case, the division result was truncated. Be sure to delete this test line after you are done playing with it.

Adding comments with

It is often useful to add notes within your programs to remind yourself, as well as inform others, how they are meant to work. These comments are indicated with a hash mark (#), also called a pound sign. Any text following this symbol to the end of the line is simply ignored by Python. The hash mark can come at the beginning of a line, or it can occur within a line after a code statement. If it occurs within quote marks, it will be interpreted as part of a string, but otherwise it signals the start of a comment.

A large block of comments is often placed at the beginning of a program, after the shebang line, to describe what the program does and how it is used. Other comments describe what major subsections of the program do, and particularly complex lines may be individually annotated for clarity. Most editors have a Comment/Uncomment Lines command which can insert # at the beginning of each line of a selection of text. You can also comment a block of lines with triple quotes (`"""`) placed before and after the block. This can be more convenient than placing a # before each line.

Commenting also serves an important function in debugging. Debugging is the process of finding and eliminating errors in your script. This process is described in detail in Chapter 13, but it is good to begin thinking about it as soon as you begin writing programs. (You won't have a choice but to think of it if things don't work as expected.) Comments can be used in debugging to effectively turn off certain portions of code, thereby helping to isolate problems. Another trick is to insert extra print statements which report on the status of your program as it executes; you can comment these extra statements out when they are not being used.

The example code in this book will include comments from this point on. As you follow along, you can insert your own comments in the code you are writing,

with observations of how and why the various portions of the programs function. In general, comments are appreciated by anyone who reads your code—even your future self when you try to revisit an old program. It is hard to have too many comments.

Controlling string formatting with the % operator

Although you have a functioning program, it is less than ideal for the task at hand. It would be clearer to present the fractions as percentages, and display the percentages with a fixed number of significant digits. The first problem just requires multiplying the fractions by 100. Controlling the number of significant digits displayed requires an additional way to format text when printing. You have already seen how to use the default behavior of the `print` command to display strings and numbers separated by commas, and how to create custom strings from mixed data types using the plus sign and the `str()` function.

A more flexible way to build strings from different data types is with the string formatting operator, `%`. It inserts the values present on the right side of the `%` symbol into the positions marked by placeholders in the strings to the left. Placeholders indicate whether the substitution should be interpreted as an integer digit (`%d`), a floating point number (`%f`), or a string (`%s`). The placeholder symbols also control the type conversion, so you don't need to use the `str()` function to convert numbers before placing them into the string. This will be much clearer with an example:

```
print "There are %d A bases." % (NumberA)
```

```
There are 3 A bases.
```

The value of the integer `NumberA` to the right of the lone `%` is inserted in the position held by `%d` in the string. Multiple values separated by commas can be simultaneously substituted into a series of placeholders, and they are substituted in the order that they occur:

```
print "A occurs in %d bases out of %d." % (NumberA, SeqLength)
```

```
A occurs in 3 bases out of 6.
```

One of the most common uses of this operator is to control the number of decimal places that are printed, using the floating point placeholder modified by the insertion of a decimal precision specifier. This consists of a dot followed by a number indicating the number of decimal digits to display, nested within the `%f`:

```
print "A occurs in %.2f of %d bases." % (NumberA/SeqLength, SeqLength)
```

```
A occurs in 0.50 of 6 bases.
```

The `%.2f` placeholder inserts the corresponding float into the string, but only with two digits of precision to the right of the decimal point. Values inserted in this way are rounded, not truncated.

Now put this into practical use in your `dnacalc.py` program. Modify your `print` statements with `%` operators as follows:

```
print "A: %.1f" % (100 * NumberA / SeqLength)
print "C: %.1f" % (100 * NumberC / SeqLength)
print "G: %.1f" % (100 * NumberG / SeqLength)
print "T: %.1f" % (100 * NumberT / SeqLength)
```

When you run the program now, the value in parentheses is calculated and inserted at the position of `%.1f`, giving you the following output:

```
host:scripts lucy$ dnacalc.py
 Sequence: ATGAAC
 Sequence Length: 6.0
A: 50.0
C: 16.7
G: 16.7
T: 16.7
```

Other ways to control the output of the `%` operator are listed in Table 8.1. These are used across many programming languages, including MATLAB, C, and Perl. There is also a `%s` placeholder, which places a string (specified either with quotes or with a variable name) within the string being formatted.

The `%` operator doesn't need to be used in conjunction with the `print` function. The formatted string could be assigned to a variable, for example:

```
pctA = "%.1f" % (100*NumberA/SeqLength)
```

With the padding feature (triggered by inserting a number after the `%`, as in `%2d` or `%4f`) you can create and print strings which are aligned along their right edges, no matter what the number of digits involved in the value. So using `"A:%3d"` would return output as follows, where each number occupies three spaces:

```
A:  2
A: 10
A:100
```

When using the `%` operator, if you actually want a percent character to show up in your string (as we might here), you have to escape your code with two consecutive percents: `'%%'`. The backslash character does not work as an escape character in this context, so `'\%'` doesn't display a percent sign.

TABLE 8.1 String formatting options

Given the string `s = '%x' % (4.13)` where `%x` is a placeholder listed below:

Placeholder	Type	Result
`%d`	Integer digits	`'4'`
`%f`	Floating point	`'4.130000'`
`%.2f`	Float with precision of 2 decimal points	`'4.13'`
`%5d`	Integer padded to at least 5 spaces	`'    4'`
`%5.1f`	Float with one decimal padded to at least 5 total spaces (includes decimal point)	`' 4.1'`

Play with the program you've written to change the way the text is displayed. The example file `dnacalc1.py` includes the version of the code discussed above.

Getting input from the user

Gathering user input with `raw_input()`

Using this example script, you can analyze different sequences by pasting them into the program file and re-running the file each time. Later you will learn to read data from a file and perform calculations on them. Sometimes, however, it is useful to have a utility program which performs calculations directly from user input—that is, from what you type at the command line. To add this feature to our example script, you can use Python's `raw_input()` function. Add this line immediately below the existing `DNASeq` definition:

```
DNASeq = raw_input("Enter a DNA sequence: ")
```

This function presents a prompt to the user, consisting of the string within the parentheses. When the user types a response and presses return, the response can then be assigned to a variable. In this case, it is assigned to the `DNASeq` variable, writing over any previous value assigned to it. The original value is forgotten.

Now when the program is run, it will wait to receive a typed or pasted value. Try different sequences to see how it works. You no longer need to edit the program to change the sequence that it considers. In Chapter 11 you will see other ways to gather input when a program is run, using the `sys.argv` function.

Sanitizing variables with `.replace()` *and* `.upper()`

Before doing anything with user input, it is a good idea to check and make sure the values entered are appropriate. As it is written, your program will count the number of uppercase `A`'s but not lowercase `a`'s. This could lead the program to

behave in ways that aren't expected by the user, who might not know that it is case-sensitive. Instead of placing the onus on the user to keep track of this, it is better to design the program so that it counts both uppercase and lowercase letters. Rather than count uppercase and lowercase letters separately, it is easier just to convert the entire string to uppercase before analyzing it. If it is all uppercase already, this will do nothing. If all or some of it is lowercase, it will change the offending characters to uppercase.

This can be accomplished using the built-in string function `.upper()`. When this method is called for a string, it returns an uppercase version of that string. This function doesn't require any parameters in the parentheses. The data it uses are in the string it is part of (that is, the string just before the period). However, since it is a function, you do need to include parentheses, even though in this case they are empty. Below your raw input, add this line:

```
DNASeq = DNASeq.upper()
```

Note that in this line of code, you call the method `.upper()` on the string `DNASeq`, and then immediately rewrite the contents of `DNASeq` with the method's result. It is usually okay to access and rewrite the same variable in one statement like this. It is necessary to do so anyway in this case, because `.upper()` alone doesn't make the string uppercase, but merely returns a new uppercase string. You can assign this new string to a new variable, or if you won't be needing the original variable again, you can overwrite it with the modification, as you did here. Rewriting variables can be a good strategy to avoid creating unneeded new variables in your programs.

Case isn't the only thing that might be a problem with our user input, however. It might also be pasted from another document and could therefore contain spaces. These spaces will not affect the count of nucleotides, but they *will* alter the total calculated length of the DNA sequence. This would result in incorrect calculations. Within a string like that generated by `raw_input()`, spaces are just another character.

To remove the spaces, you can use another built-in string function: `.replace()`. This function takes two arguments separated by commas: the item to be removed, and the item to replace it with. In this case, we will remove a space, `" "`, and replace it with an empty set of quotes, `""`. Below the `.upper()` command, insert this `.replace()` function:

```
DNASeq = DNASeq.replace(" ","")
```

This is just a simple search and replace, not a regular expression. You will learn how to use regular expressions within a Python program a bit later. Also note that we again had to reassign the result of this function to the variable `DNASeq`, since it doesn't directly search and replace within the original string, but instead generates a new string.[4]

Now you have a moderately sanitized version of the `DNASeq` string, ready to work with the rest of the program you already wrote. Before sending your program out into the real world for others to use, you would want to incorporate further checks—for instance, making sure the resultant string length is not zero, and that there are only legal characters in it (`C`, `G`, `A`, `T`). For the moment, though, you have a nice utility for performing calculations based on the character composition of a string. This same kind of functionality can be used to make a great many useful calculations, such as the molecular weight of a protein or molecule, or the melting temperatures of oligonucleotides. Here is the final script:

```
#!/usr/bin/env python
# This program takes a DNA sequence (without checking)
# and shows its length and the nucleotide composition

DNASeq = "ATGAAC"
DNASeq = raw_input("Enter a DNA sequence: ")
DNASeq = DNASeq.upper()  # convert to uppercase for .count() function
DNASeq = DNASeq.replace(" ","")  # remove spaces

print 'Sequence:', DNASeq

SeqLength = float(len(DNASeq))

print "Sequence Length:", SeqLength

NumberA = DNASeq.count('A')
NumberC = DNASeq.count('C')
NumberG = DNASeq.count('G')
NumberT = DNASeq.count('T')

# Old way to output the Numbers, now removed
# print "A:", NumberA/SeqLength
# print "C:", NumberC/SeqLength
# print "G:", NumberG/SeqLength
# print "T:", NumberT/SeqLength

# Calculate percentage and output to 1 decimal
print "A: %.1f" % (100 * NumberA / SeqLength)
print "C: %.1f" % (100 * NumberC / SeqLength)
print "G: %.1f" % (100 * NumberG / SeqLength)
print "T: %.1f" % (100 * NumberT / SeqLength)
```

[4] You can also nest methods by putting consecutive dot notation on the same line: `DNASeq.upper().replace(" ","")`

Reflecting on your program

The program you've constructed could easily be modified to be a general calculator applied to other purposes. As we mentioned, though, it is not written in a very optimal way, because instead of using a loop to do the repetitive commands, we have just duplicated the `.count()` and `print` lines and manually edited them. This is officially Bad Programming Practice. In the next chapter you will see a simpler way to work through the elements of a string or a list of variables using a `for` loop. At the end of Chapter 9, we will revisit this program and show how to accomplish most of its capability in a script of only four lines.

SUMMARY

You have learned how to:

- Create a Python program and execute it
- Use the function `len()`
- Change variable types with `int()`, `float()`, and `str()`
- Add comments to your program with #
- Print strings and numbers together
- Format strings to your own specification with `%d`, `%.2f`, and `%s`
- Get user input with `raw_input()`
- Use the built-in string functions `.count()`, `.replace()`, and `.upper()`

Chapter 9

DECISIONS AND LOOPS

You now are able to construct and run a basic program in Python starting from only a blank text file. This chapter takes a brief break from writing programs to show the interactive prompt—an area that functions like a scratchpad, for testing small bits of code—and then describes finding help for different elements of the Python language. You will then augment the program you wrote in the previous chapter so that it can make decisions and work with lists of data. You will also write a protein calculator, which requires converting a web-formatted table so it can be used by your program. Finally, the chapter will explain lists in detail, so that you become comfortable working with this important type of variable.

The Python interactive prompt

Sometimes you just want to quickly try out a few different Python commands to see what they do. You could write a whole program that just contains those few lines, but most of your time would go into getting the file set up to run. As an alternative, you can use the Python programming language through an **interactive prompt**. This interface allows you to try Python code right at the command line without ever generating a program file. We often work with one terminal window open to the shell for executing programs, and a second terminal window open to a Python prompt to test pieces of code and check results before we place them into the actual program file.

INTERACTIVE PROMPTS VERSUS EXECUTABLE TEXT FILES Using a program interactively versus sending it a series of commands in a text file is not as different as might seem. When you execute a text file containing a shell script or a Python program, the shebang line at the very start of the file instructs the system to send the rest of the contents of the file to the appropriate program (in this case, `bash` or `python`). For the most part, the programs don't even know whether the commands they are running are coming from the user or from a file.

To launch the Python interactive prompt, open a new terminal window and type `python` at the shell prompt:

```
host:~ lucy$ python
Python 2.6.2 (r262:71600, Apr 16 2010, 09:17:39)
[GCC 4.0.1 (Apple Computer, Inc. build 5250)] on Darwin
Type "help", "copyright","credits" or "license" for more information.
>>>
```

You will see some diagnostic information, including the Python version number. If the version reported is between 2.3 and 2.7, you should be okay, although some specific commands may differ slightly in versions older than 2.5. After the diagnostic information has been printed, the prompt then changes to >>>, indicating you are now in Python's interactive mode.

Try the following Python commands at the interactive prompt to get a feel for how it works. If you initialize a variable with a value, then enter the variable name either by itself or as part of an operation, the resulting value is printed to the screen:

```
>>> x='53'  ← No output is generated; the variable x is initialized with a string
>>> x
'53'  ← The value of the variable x is printed to the screen
>>> x+2
Traceback (most recent call last):  ← Fails because of type mismatch
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' object
>>> float(x)+2
55.0  ← Note the decimal indicating the result is a float; the value of x remains unchanged
>>> int(x)/2
26  ← Note lack of decimal since the result is an integer
```

One useful feature of the Python prompt is the ability to get information about the variables in your workspace. The `dir()` function lists all the variables and methods that are nested within a specified variable. If you run `dir()` on a string,

you will see some familiar functions, such as `.replace()` and `.upper()`. You will also see some names that begin and end with __, but these are internal names that we have trimmed from this screen capture:

```
>>> DNASeq='ATGCAC'
>>> dir(DNASeq)
['capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'index', 'isalnum',
'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

In this case, we have defined the string called `DNASeq`, and then run `dir()` to see all of its components. These are the methods and variables nested within a string. Any of these can be accessed using dot notation:

```
>>> DNASeq.isdigit()
False
>>> DNASeq.lower()
'atgcac'
>>> b=DNASeq.startswith('ATG')
>>> print b
True
>>> DNASeq.endswith('cac')  ← Case-sensitive
False
```

A variation of the `dir()` command is to use the `help()` function on a variable. If you type `help(DNASeq)`, it will print out an extended version of the commands that are available for this variable type.This doesn't work for all variables and for older versions of Python, so you might have to enter the name of the variable *type* instead, as in `help(list)` or `help(str)`.

You can also use the Python interpreter to check your program: just copy and paste portions of a script into the window, press return, and it will attempt to run. (This works best when the script does not involve reading files or asking for user input.) The variables will be loaded into memory so that you can interact with them, to check that things are operating as you expect. As you will see shortly, indentation makes a difference in Python, so you have to pay attention to both extra space and missing space at the beginning of the lines that you paste into the interactive prompt. Use ctrl Z.

When you are done with the Python interpreter, you can return to the `bash` shell prompt by typing `quit()` or ctrl D.

Getting Python help

The built-in Python documentation has many shortcomings. To begin with, it isn't installed by default on many computer systems. Even if it is present, there are still problems. The documentation is difficult to search, and once you find a promising document, the majority of the document typically explains the history of the command and why it was constructed as it is, rather than explaining what it does and showing examples of how to use it. For this reason we are reluctant to recommend the `help()` function at the Python prompt, or even the `pydoc` program available at the `bash` command line.

As with most things now, the quickest way to answer your Python-related question is through a Web search. This doesn't help if you are on an airplane or sitting at the beach, but in general it will give you the fastest results. You can often do a general Web search including the word `python` (maybe also `-monty -snake` to remove spurious results), plus the concept you are interested in, and get your question answered. Some helpful sites dedicated to Python include:

```
http://python.about.com/
http://docs.python.org/tutorial/
http://rgruet.free.fr/PQR26/PQR2.6_modern_a4.pdf
http://rgruet.free.fr/PQR25/PQR2.5.html
http://www.diveintopython.org/
```

The Python commands discussed in this book are summarized in Appendix 4 for easy reference, and Chapter 13 offers help on debugging.

Adding more calculations to `dnacalc.py`

You will now continue building on the `dnacalc.py` script from the previous chapter. The next challenge is to estimate the melting temperature for the sequence (the temperature at which the strands of a DNA molecule with this sequence would separate from their complement, in half of the sample). This calculation will be based on the counts of various nucleotides in the sequence, and you will employ a different formula depending on the overall length of the sequence. To get started again, reopen your `dnacalc.py` script in TextWrangler, or open `dnacalc1.py` from the `pcfb/examples` folder.

Although the ability to accept user input at the command line makes your program more convenient to use, during the development process it is usually faster to hardcode a fixed value for testing. Find the `raw_input` statement around line 10 and comment it out by adding # at the beginning of the line. This turns off the input statement and retains the hardcoded value for the `DNASeq` variable in the preceding line:

```
DNASeq = "ATGTCTCATTCAAAGCA"  ← Now using a longer sequence
# DNASeq = raw_input("Enter a sequence: ")  ← Not executed anymore
```


Now you can add the ability to calculate the melting temperature, also known as Tm. One simple approximation for Tm is based on the number of strongly binding nucleotide pairs (G+C) and the number of weak ones (A+T). At the end of your existing program, add these lines to calculate the Tm:

```
TotalStrong = NumberG + NumberC
TotalWeak = NumberA + NumberT
MeltTemp = (4 * TotalStrong) + (2 * TotalWeak)
print "Melting Temp : %.1f C" % (MeltTemp)
```

Run the program and see that the result of the new calculation is displayed.

Conditional statements with `if`

In reality, this formula for the melting temperature is only recommended for short nucleotide sequences. If sequences are 14 or more nucleotides long, you should use another formula rather than the one shown above.[1]

To implement this "conditional behavior" where different formulas are employed under different circumstances, you will use the `if` statement. The `if` statement in Python has this general syntax (shown in pseudocode[2]):

```
if logical condition == True:  ← Notice the colon
    do this command  ← Indented block
    do this command
continue with normal commands  ← Main thread of the program
```

The commands that are indented under the `if` statement constitute what is sometimes called a **block** of code—that is, the commands are not only contiguous, but constitute a functional unit, with the entire set of lines executed one after another. In this case, this particular block is only executed if the condition is logically true. Otherwise, if the condition is false, the block is skipped and the program continues executing below.

Designating code blocks using indentation

The method of indicating which commands are controlled by the `if` statement brings us to one of the most unique aspects of Python: *indentation is used to indicate blocks of code*. The programs you have written so far have consisted of only a single block of code, run from top to bottom. From here on out, though, this strictly linear sequence of events will often be modified with conditional statements that only execute some blocks of code if certain criteria are met, as well as with blocks of code whose commands are reiterated multiple times within a loop. In each case, blocks are nested within other blocks. Therefore you need to be able to signal

[1] According to `http://www.promega.com/biomath/calc11.htm`.

[2] Pseudocode is a combination of computer code and plain English, used for illustrative purposes.

which groups of lines go together, forming a block, and thus indicate how the flow of the program will proceed.

In many other programming languages, nested blocks of code must be designated with brackets, with indentation being technically superfluous. Even in these other languages, however, programmers typically indent blocks anyway, to improve the readability of their code. Just remember that in Python, indentation of nested blocks is not an option or an aesthetic decision, but a requirement.

Indentation can be produced by either tabs or spaces. If you choose to use spaces, you must always use the same number of spaces to designate the same block. (Each block is typically indented four spaces relative to the block it is nested within.) We find it easier to just stick with tabs, and so they are used throughout this book and in the sample files. When copying and pasting code from programs you find on the Web, be prepared for inconsistencies in indentation. One program might use tabs, another three spaces, another four spaces. Web examples can also contain invisible characters that are not normal spaces. This can lead to strange behavior if you combine code of disparate origins into one program. It is important to go back through all the copied code and make sure the indentation characters are consistent, since in most environments, you can't mix tabs and spaces, nor different numbers of spaces. In TextWrangler, you can turn on Show Invisibles to see such characters in your document.

This formatting via visual indentation corresponds to the flow of the program. For example, the flow of a program takes a detour when it reaches an `if` statement with conditions that are true, executes the indented block of code, and then continues on. If the condition is false, the detour is avoided and the program goes on as if the indented code didn't even exist.

The line leading to an indented block of code ends with a colon, whether the line is an `if` statement, as described here, or controls a loop, which we will discuss a bit later. Unlike other languages, there is no specific marker at the end of a block of code in Python. The code present where normal operation resumes after the end of a nested block is aligned with the indentation of whatever statement started the block, be it an `if` statement or some other statement.

TABLE 9.1 Python comparison operators

Comparison operators These operators return `True` (`1`) or `False` (`0`) based on the result of the comparison.

Comparison	Is `True` if...
`x == y`	x is equal to y
`x != y`	x is not equal y
`x > y`	x is greater than y
`x < y`	x is less than y
`x >= y`	x is greater than or equal to y
`x <= y`	x is less than or equal to y

Logical operators

Conditional statements used in `if` statements and throughout programming are usually either simple comparisons such as `SeqLength >= 14`, tests of equality such as `SeqType == 'DNA'`, or else some logical combination of these, such as `(Latitude > 30 and Latitude < 40)`. This last example can also be written as `30 < Latitude < 40`.

Be sure that when comparing the equality of two entities, you use two consecutive equal signs (==) as the operator. A single equal sign would assign the value of the second entity to the first. This is a common slipup when first programming. See Tables 9.1 and 9.2 for comparison and logical operators in Python.

TABLE 9.2 Python logical operators

Logical operators In this table, `A` and `B` represent a true/false comparison like those listed Table 9.1.

Logical operator	Is `True` if...
`A and B`	Both `A` and `B` are true
`A or B`	Either `A` or `B` is true
`not B`	`B` is false (inverts the value of `B`)
`(not A) or B`	`A` is false or `B` is true
`not (A or B)`	`A` and `B` are both false

The `if` statement

To put all this into practice, you will add an `if` statement that will execute one block of code if the sequence is 14 or more characters long, and another block of code if the sequence is less than 14 characters long. In the final version of this program, one block or the other, but never both blocks, will be run.

Insert these lines in the program just above where the `MeltTemp` is calculated:

```
if SeqLength >= 14:
    #formula for sequences 14 or more nucleotides long
    MeltTempLong = 64.9 + 41 * (TotalStrong - 16.4) / SeqLength
    print "Tm Long (>14): %.1f C" % (MeltTempLong)
```

You can either type this or copy it from the `dnacalc1.py` example file.

Go through these new commands line-by-line. First, the `if` statement itself is just a test of whether the `SeqLength` variable is 14 or greater. If this condition is true, it executes the next two indented lines and then goes on to also complete the Tm calculation designed for short DNA sequences. If it is false, the program skips the calculation for long sequences and goes straight to the Tm calculation designed for short DNA sequences.

The `else:` statement

As the program stands, the Tm derived from the short-sequence formula will be printed no matter what the sequence length is. This is a bit annoying, since the number resulting from the short calculation is irrelevant if the sequence is 14 or more nucleotides in length. In many situations where you are using the if statement, including this one, you want one block of commands to be executed *only* if the condition is `True`, and another *only* if it is `False`. To accomplish this, use the `else:` statement, which provides an alternative route, or detour, to be executed only when the main condition is `False`.

Add an `else:` statement immediately before the formula for sequences less than 14 nucleotides long. Then indent the `MeltTemp` calculation line and the `print` statements so that they form a nested block of commands under the control of the `else:` statement. Remember, even though it is just a single word by itself,

an `else:` statement must always have a colon at the end of the line, just like an `if` statement.

In TextWrangler, you can easily indent existing lines by highlighting them and pressing ⌘]. Similarly, ⌘[moves a block of highlighted commands to the left. The entire modified program should now look like this:

```
#! /usr/bin/env python
# This program takes a DNA sequence (without checking)
# and shows its length and the nucleotide composition

DNASeq = "ATGTCTCATTCAAAGCA"
# DNASeq = raw_input("Enter a sequence: ")
DNASeq = DNASeq.upper()  # convert to uppercase for .count() function
DNASeq = DNASeq.replace(" ","") # remove spaces

print 'Sequence:', DNASeq

# below are nested functions: first find the length, then make it float
SeqLength = float(len(DNASeq))

# SeqLength = (len(DNASeq))
print "Sequence Length:", SeqLength

NumberA = DNASeq.count('A')
NumberC = DNASeq.count('C')
NumberG = DNASeq.count('G')
NumberT = DNASeq.count('T')
# Calculate percentage and output to 1 decimal
print "A: %.1f" % (100 * NumberA / SeqLength)
print "C: %.1f" % (100 * NumberC / SeqLength)
print "G: %.1f" % (100 * NumberG / SeqLength)
print "T: %.1f" % (100 * NumberT / SeqLength)

# Calculating primer melting points with different formulas by length

TotalStrong = NumberG + NumberC
TotalWeak = NumberA + NumberT

if SeqLength >= 14:
   #formula for sequences > 14 nucleotides long
   MeltTempLong = 64.9 + 41 * (TotalStrong - 16.4) / SeqLength
   print "Tm Long (>14): %.1f C" % (MeltTempLong)
else:
   #formula for sequences less than 14 nucleotides long
   MeltTemp = (4 * TotalStrong) + (2 * TotalWeak)
   print "Tm Short: %.1f C" % (MeltTemp)
```


Now you can assign DNA sequences of varying lengths to the `DNASeq` variable and test the effectiveness of your conditional expressions. You can also reinstate the `raw_input` command, to once again make the program work interactively.

Choosing from many options with `elif` When you are using combinations of `if` and `else:` to choose from a long list of items, you can get into a deeply indented series of statements:

```
if X == 1:
    Value = "one"
else:
    if X == 2:
        Value = "two"
    else:
        if X == 3:
            Value = "three"
```

A cleaner way to write this is by using the special `elif` command, short for **else if**:

```
if X == 1:
    Value = "one"
elif X == 2:
    Value = "two"
elif X == 3:
    Value = "three"
```

In this trivial example, an even better way to solve the problem would be to think like a programmer and use a list, with `X` as an index to the list, rather than as part of a logical test. You will learn about lists later in this chapter:

```
ValueList=["zero","one","two","three"]
Value = ValueList[X]
```

Introducing `for` loops

In the next section, you will create a simple script called `proteincalc.py`. This program will take an amino acid sequence as input. Amino acids and their corresponding molecular weights will be stored in a dictionary. You will use a `for` loop to calculate the total molecular weight of the protein by adding up the molecular weight of each of its amino acids.

Remember from Chapter 7 the basic premise of a `for` loop: it cycles through each object in a list or other collection of variables, performing a block of commands each time through. In Python pseudocode the syntax is similar to an `if` statement:

```
for MyItem in MyCollection:
    do a command with MyItem
    do another command
# return to the for statement and move to the next item
resume operation of main commands
```

This `for` statement says: "For each item in `MyCollection`, assign that value to `MyItem`, and run the block of commands below. Then go through the next item in `MyCollection` and run the commands again."

The first time through the loop, the variable `MyItem` temporarily takes on the value of the first item in `MyCollection`. Within the loop, any calculations or print operations that involve `MyItem` will be done using this first value. When the last statement of the indented block is complete, the execution returns to the `for` line, and `MyItem` takes on the second value of `MyCollection`. Finally, after `MyItem` has been assigned the last value and the last of the indented statements has been executed, the program continues with the following unindented line.

A brief mention of lists

Items in `MyCollection` are typically of the type list. In Python, lists are created with square brackets enclosing a group of items, and with each item separated by commas:

```
MyCollection=[0,1,2,3] ← List of integers
MyCollection=['Deiopea','Kiyohimea','Eurhamphaea'] ← List of strings
MyCollection=[ [34.5, -120.6] , [33.8, -122.7] ]  ← List of lists
```

In the last example, where `MyCollection` is a list of lists, in the course of cycling through the `for` loop, `MyItem` itself would be a two-element list, so that a pair of values could be used within the loop.

There is also a built-in Python function called `list()` which can convert the contents of a string into list elements. For example:

```
MyCollection= list('ATGC')
```

is the same as:

```
MyCollection = ['A', 'T', 'G', 'C']
```

This command can save you a lot of typing of commas and quotation marks, but in the context of a `for` loop, unless you want to sort the list, such conversions are not necessary: in Python, `for` loops can operate directly on each character of a string without conversion to a list.

Writing the `for` loop in `proteincalc.py`

Begin writing a new script called `proteincalc.py` in the editor. (The completed script is included in the examples folder as well.) Prepare your script for execution by performing the now-routine steps of adding a `#!` line at the top, saving the file to your scripts folder, and doing a `chmod u+x` to make it executable. Then add a string definition and the lines of code below:

```
#!/ usr/bin/env python
ProteinSeq = "FDILSATFTYGNR"
for AminoAcid in ProteinSeq:
    print AminoAcid
```

Now try running your program. If you get an error, make sure that you remembered the colon after `ProteinSeq` at the end of the `for` statement. Notice that this loop treats the variable `ProteinSeq` as a list: it steps through each letter of the string, assigns it to `AminoAcid` in turn, and prints each on its own line. So the first time through the loop, it is as if there is a statement saying `AminoAcid = 'F'` before the `print` statement.

For quick debugging, many text editors have a tool to execute code without switching to the terminal window, or they can automatically switch to the terminal for you. In TextWrangler, the menu with this option is easy to miss since it is labeled with a shebang (`#!`).

One potentially confusing aspect of `for` loops is that variables like `AminoAcid` are never mentioned or defined before appearing in the `for` statement. This is because they are defined each time the `for` statement is executed.

Now that you have a way to cycle through each amino acid in the protein sequence (each character in the string), you will use a Python dictionary along with that bit of information to look up a corresponding value in a table of molecular weights.

Generating dictionaries

Recall from Chapter 7 that dictionaries are collections of objects, with the objects being looked up by associated keys, rather than by order of occurrence in the collection. In our example, we want to associate the amino acid's single letter code (such as `'A'`) to its corresponding molecular weight (a floating point value such as `89.09`).

There are several ways to create dictionaries. One is by defining the pairs of keys and values inside curly brackets. Keys and their values are separated by colons, and each pair is separated by a comma, with the basic format of:

```
MyDictionary = {key:value, nextkey:nextvalue}
```

For example:

```
TaxonGroup={'Lilyopsis':3, 'Physalia':1, 'Nanomia':2, 'Gymnopraia':3}
```

In this example, the keys are strings, and the values are integers.

Although dictionaries are *created* using `{}`, values are *extracted* from the dictionary using square brackets `[]`, which is the same way that values are extracted from lists. So given the `taxongroup` dictionary above, the command:

```
print TaxonGroup['Lilyopsis']
```

would print out the value 3.

OTHER WAYS TO CREATE DICTIONARIES There are several other ways to create dictionaries. Which method you use depends on the format of the information you have to begin with. In addition to the `{}` method described in the text, another useful method is when you have two lists of equal length, one containing keys and the other containing values; in such a case, you can use the `dict()` and `zip()` functions to associate them. For example, if you have these lists:

```
TaxonKeys=['Lilyopsis', 'Physalia', 'Nanomia', 'Gymnopraia']
TaxonValues=[3, 1, 2, 3]
```

you can create an association between them using the command:

```
TaxonGroup=dict(zip(TaxonKeys,TaxonValues))
```

First the `zip` command pairs the two lists together into a list of pairs, and then the `dict` command takes these pairs and connects them as dictionary entries.

Despite its strict indentation rules, Python does allow a statement to be split across lines if the splits occurs within `()`, `[]`, or `{}`. This is often useful for large entries, such as long lists or dictionaries. As an example, the definition above could also be written as:

```
TaxonGroup={
'Lilyopsis' :3,
'Physalia'  :1,
'Nanomia'   :2,
'Gymnopraia':3 }
```

For your `proteincalc` program, you will be making a comparable dictionary for amino acids. We won't ask you to type out a dictionary definition of twenty amino acids and their molecular weights. This kind of busywork wastes time and also has a high probability of introducing errors. Instead, you will use your regular-expressions skills to convert data derived from a web page into Python code.

It will often be convenient for you to gather tables of information from the Web, other spreadsheets, or documents you already have. In all likelihood, these data will not have been intended for use in a computer program, but converting them into programming syntax is usually a matter of doing a few searches and replacements.

In this case, you will create a dictionary where each line has an amino acid as the key and its molecular weight as the value:


```
AminoDict={
'A':89.09,
'R':174.20,
...several more lines...
'X':0,
'-':0,
'*':0 }
```

Drag the file `aminoacid.html` from your `examples` folder into a web browser, or obtain the file from `http://practicalcomputing.org/aminoacid.html`. You should see a table of amino acid names, abbreviations, and molecular weights (Figure 9.1).

The first step is to get these data into a blank text document so that you can edit them. One way to do this would be to copy and paste them from the web page. This would work fine in many situations. However, tables copied from some browsers end up as lists rather than tables when they are pasted into a text document. There might also be hidden characters on the Web site that you don't see, but which can cause problems in the text file.

You will usually get more consistent results when you pull web data directly from the page source code. You can view the source code of any Web page by selecting Page Source or View Source from your browser's View menu. (The exact wording and location of this command will depend on your browser, but with a bit of digging you should be able to find it. You can also right-click on the page and see if there is an option to view the source.) Take a look to see where the data of interest start and end in the file. Even if you don't understand a word of HTML (the language used to encode most web pages) you will quickly be able to identify the letter code and molecular weight of each amino acid within this text. Most web pages will have more extraneous text than this one, but the essential information will be embedded somewhere within.

Name	Abbreviation	Single-Letter	Mol Wt
Alanine	Ala	A	89.09
Arginine	Arg	R	174.20
Asparagine	Asn	N	132.12
Aspartic acid	Asp	D	133.10
Cysteine	Cys	C	121.15
Glutamine	Gln	Q	146.15
Glutamic acid	Glu	E	147.13
Glycine	Gly	G	75.07
Histidine	His	H	155.16
Isoleucine	Ile	I	131.17
Leucine	Leu	L	131.17
Lysine	Lys	K	146.19
Methionine	Met	M	149.21
Phenylalanine	Phe	F	165.19
Proline	Pro	P	115.13
Serine	Ser	S	105.09
Threonine	Thr	T	119.12
Tryptophan	Trp	W	204.23
Tyrosine	Tyr	Y	181.19
Valine	Val	V	117.15
Unknown	Xaa	X	0
Gap	Gap	-	0
Stop	End	*	0

FIGURE 9.1 The table of amino acid names, abbreviations and molecular weights contained in `aminoacid.html`

In this case, the useful information is all in lines that start with `<tr><td>`, signifying the start of a table row boundary. Every line that starts with `<tr><td>`, except the header line that starts `<tr><td>Name`, contains the properties and name of an amino acid, as it would be indicated in a protein sequence. Copy these lines, from the one that begins with `<tr><td>Alanine` through the one that begins with `<tr><td>Stop`, and paste them into a blank text file in your text-editing program.

The data will look like this:[3]

```
<tr><td>Alanine</td><td>Ala</td><td>A</td><td>89.09</td></tr>
<tr><td>Arginine</td><td>Arg</td><td>R</td><td>174.20</td></tr>
<tr><td>Asparagine</td><td>Asn</td><td>N</td><td>132.12</td></tr>
...omitted lines...
<tr><td>Unknown</td><td>Xaa</td><td>X</td><td>0.0</td></tr>
<tr><td>Gap</td><td>Gap</td><td>-</td><td>0.0</td></tr>
<tr><td>Stop</td><td>End</td><td>*</td><td>0.0</td></tr>
```

Remember your goal is to reformat those four text fields so they look like this:

```
'A':89.09,
'R':174.20,
```

You will now use regular expressions to reformat each of these lines into Python code that you can copy and paste into your program file. (If you don't recall how to use regular expressions, turn back to Chapters 2 and 3). Look through the raw data to find the minimal chunk of text that has all the information you need. In the first line this would be:

```
A</td><td>89.09
```

This is the single-character code that would be used in a protein sequence (`A`, `R`, to `-` and `*`), followed by a few formatting characters that are the same in every line, followed by the molecular weight value.

The first step in constructing the regular expression is to figure out the search term. To capture both fields of interest, you can use:

```
.+(.)</td><td>([\d\.]+).+
```

This search term has the following components:

`.+` stands for any series of one or more characters

`.` stands for any single character by itself, which you want to capture

`()` mark the portions of the search you want to capture

`</td><td>` is the separating text, which is the same on each line

`[\d\.]+` is any series of one or more digits and decimal points

The reason we use a dot in this search instead of `\w` for the first character we are capturing is that there are some symbols at the bottom of the list which are not letters or numbers. The leading and trailing `.+` are necessary because we want the search term to match the entire line, not just the part of interest, so that this flanking text is removed in the search and replace. This matches only the correct portion of the line because there is only one place where the numbers occur.

[3]To get colored formatting, tell TextWrangler that this is an XML or HTML file using the pop-up menu at the bottom of the page. (It will say None by default.)

For a replacement string, use:

```
'\1':\2,
```

This will put the first bit of captured text, designated by `\1`, between a pair of single quotation marks. It will then add a colon and the second bit of captured text.

Execute the search and replace, and you should get the following:

```
'A':89.09,
'R':174.20,
'N':132.12,
...omitted lines...
'X':0.0,
'-':0.0,
'*':0.0,
```

To finish off the dictionary statement, add a line above this list which begins the definition, then delete the comma after the final item in the list, and finish it off with a closing curly bracket as follows:

```
AminoDict={
'A':89.09,
'R':174.20,
'N':132.12,
...omitted lines...
'V':117.15,
'X':0.0,
'-':0.0,
'*':0.0
}
```

Your text file now contains legal Python code which is ready to be copied and pasted into your `proteincalc.py` script. Paste these lines into your existing script just above the definition of the variable `ProteinSeq`.

This dictionary will let your program look up values using the single-letter amino acid name in square brackets. For example, to use the molecular weight of methionine in your script, you can type `AminoDict['M']` and it will give you the corresponding numerical value. As an intermediate test, you could modify the current `print` statement so that it prints both the name and the associated value:

```
print AminoAcid, AminoDict[AminoAcid]
```

Using this dictionary and the `for` loop you have already created, you can now step through each amino acid in `ProteinSeq`, look up its value in the diction-

ary, and add it to a variable `MolWeight` to sum up the full molecular weight. Modify your existing code to replace the `print AminoAcid` statement in the loop with the running total of the `MolWeight` as shown below. Before adding to the `MolWeight` variable, you need to initialize it to zero:

```
MolWeight = 0
for AminoAcid in ProteinSeq:
    MolWeight = MolWeight + AminoDict[AminoAcid]
```

This loop takes each character of the string `ProteinSeq` and assigns its value to the `AminoAcid` variable. Using this letter as a key, it retrieves the corresponding value from the dictionary called `AminoDict`. It adds this value to the value of `MolWeight`, which grows in value each time through the loop.

Once the loop is completed, print `ProteinSeq` and the molecular weight:

```
print "Protein: ", ProteinSeq
print "Molecular weight: %.1f" % (MolWeight)
```

You could also modify this program by adding user input, so that the user is asked to provide a sequence when the program is run. If you did this, you would want to use the `.upper()` function in the same way it was used in `dnacalc.py`, to make sure the user input is in uppercase. This is important because dictionaries give an error when you try to look up a key (such as a lowercase `'a'`) which is not among the defined entries. (There is a way around this using the `.get()` function to retrieve variables, as discussed in the next section.)

The entire program is summarized below. To save space, the dictionary has been reformatted here to occupy fewer lines. It will still run in this format:

```
#! /usr/bin/env python

# This program takes a protein sequence
# and determines its molecular weight
# The look-up table is generated from a web page
# through a series of regular expression replacements

AminoDict = {
'A':89.09,   'R':174.20, 'N':132.12, 'D':133.10,
'C':121.15,  'Q':146.15, 'E':147.13, 'G':75.07,
'H':155.16,  'I':131.17, 'L':131.17, 'K':146.19,
'M':149.21,  'F':165.19, 'P':115.13, 'S':105.09,
'T':119.12,  'W':204.23, 'Y':181.19, 'V':117.15,
'X':0.0,     '-':0.0,    '*':0.0     }
```



```
# starting sequence string, on which to perform calculations
# you could use raw_input().upper() here instead
ProteinSeq = "FDILSATFTYGNR"

MolWeight = 0

# step through each character in the ProteinSeq string,
# setting the AminoAcid variable to its value
for AminoAcid in ProteinSeq:

   # look up the value corresponding to the current amino acid
   # add its value of the present amino acid to the running total
   MolWeight = MolWeight + AminoDict[AminoAcid]

# once the loop is completed, print protseq and the molecular weight
print "Protein: ", ProteinSeq
print "Molecular weight: %.1f" % (MolWeight)
```

Other dictionary functions

Given that you will probably be working extensively with dictionaries, a few other commands will be useful to you.

The `.get()` function In addition to using square brackets to extract values from a dictionary, you can use the `.get()` function. For retrieving the value associated with the key `'A'`, the equivalent statement to `AminoDict['A']` would be:

```
AminoDict.get('A')
```

This function operates just like square brackets, except that you can specify a default value to be returned if the entry doesn't exist. For example, instead of defining `0.0` as the molecular weight for stop codons (`*`) and for dashes, you could access your dictionary with the statement:

```
AminoDict.get(AminoAcid,0.0)
```

where the parameter after comma is the default value to use. Be careful if you use this formulation, because you won't get an error if your input sequence is somehow scrambled and includes improper characters or other punctuation.

Listing keys and values A list of the keys in a dictionary can be extracted using the `.keys()` function. Remember that there is no intrinsic order to the keys or values in a dictionary. You can't count on them being alphabetical, or even in the same order that they were entered. If you want to loop through each key of a dictionary in some kind of sorted order, use the `.keys()` command, along with the `sorted()` function (described at length later in this chapter) to produce a separate list:

```
SortedKeys = sorted(AminoDict.keys())
```

You can loop through this list using:

```
for MyKey in SortedKeys:
```

A list of all of the values of a dictionary can be retrieved using the `.values()` method:

```
AminoDict.values()
```

Although the keys and values will not be returned in a predictable order, the output from the `.keys()` and `.values()` methods will occur in the same order relative to each other.

Applying your looping skills

Although this program is doing something more intricate than your `dnacalc` program from earlier in the chapter, it is nonetheless accomplishing its task in fewer steps. This is because it is programmed in a more efficient and flexible manner.

You could rewrite the first part of the `dnacalc.py` program, which calculates the percentage of each nucleotide in a DNA sequence, in a similar fashion:

```
#! /usr/bin/env python
DNASeq = "ATGTCTCATTCAAAGCA"
SeqLength = float(len(DNASeq))

BaseList = "ACGT"
for Base in BaseList:
    Percent = 100 * DNASeq.count(Base) / SeqLength
    print "%s: %4.1f" % (Base,Percent)
```

This example is available as `compositioncalc1.py`. In this case, instead of looping through the sequence, we are looping through the list of bases that we want to count within the sequence. The `print` line has also been formatted with `%4.1f` to pad the output to four total spaces, producing output where the values are aligned along their right edges:

```
A: 15.4
C:  7.7
G: 30.8
T: 30.8
```

The power of this approach is something you should use frequently in your scripts. To add support for counting ambiguity codes in the sequence, like `'S'` (strong nucleotide pairs `G` or `C`) and `'W'` (weak nucleotide pairs `A` or `T`), you could just add them to the `BaseList` string, and their percentages would also be calculated and printed without adding any additional lines of code.

Lists revisited

Lists are an integral part of many programs, particularly those designed to analyze and convert large datasets. While we have introduced aspects of lists in several places, here we will take a break from developing programs, and look in more detail at ways to use lists in Python in particular, and at the related commands for doing so (see also Appendix 4).

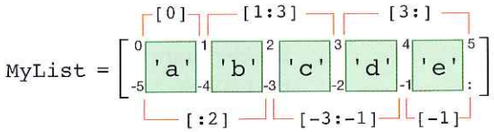

FIGURE 9.2 Numerical ways to think about indexing list elements

Indexing lists

Lists are collections of values. These collections are defined using `[ ]`, for example `MyList = ['a','b','c']`. Unlike in some languages, in Python the items in a list can be a mixture of data types, including strings, numbers, and even other lists. In addition to being defined with square brackets, list elements are also retrieved from a list using square brackets, as in `MyList[1]`. Between the brackets, there can be a single number, or a range of numbers separated by a colon, for example `MyList[1:3]`. List indexing can be confusing for several reasons: first, the index numbers start at zero, not one; second, indices sometimes don't seem to line up with their values; and third, Python indexing is handled differently than in some other programming languages. A diagram can help, in this case showing a list of 5 characters `'a'` to `'e'` (Figure 9.2).

The natural inclination is to think of the indices in brackets as corresponding to particular boxes in the list. If you want to extract `b,c`, you might think to try `[1:2]` or `[2:3]`, but neither is correct! In Python, it is better to imagine your list as a series of boxes with numbers placed *between* them. Indices can be positive, counting up from zero at the beginning of the list, or they can count back from the end of the list using negative numbers. If a single index without a colon is included within the brackets, it specifies the element to the right of the indicated boundary. A colon turns the brackets into tongs that reach into the list at particular numbered locations to grab the elements between them, such as `[1:3]`, which grabs elements `b` and `c`. Another way to think of it is that the first element in the brackets is inclusive, meaning that this element is included in the range, and the element after the colon is exclusive and is not part of the recovered range. Both ends of the range may be specified, providing for left and right boundaries within the list. If one of the numbers in a range is omitted (for example `[:3]` or

THE DUAL ROLE OF BRACKETS You have probably noticed that square brackets, `[ ]`, are used in a couple of unrelated ways with lists. First, they are used to define lists with particular elements, in much the same way that quotes are used to define strings. In the statement `MyList = [33,12,89]`, for example, the brackets are used to define a list that is then assigned to the name `MyList`. If the brackets immediately follow the list name, however, they are interpreted as specifying particular elements in the list by their indices; thus `MyList[1]` returns the second element of `MyList`, because `[1]` is not defining a list but specifying an index—in effect, a range within the list to extract.

`[2:]`), then this grabs from the beginning of the list (if the number to the left of the colon is omitted) or up to the end of the list (if the number to the right of the colon is omitted). A colon by itself within the brackets specifies all elements from the beginning to the end of the list; thus `MyList[:]` returns a full copy of all the elements in `MyList`.

Open the Python interpreter at the command line (just type `python` at the shell prompt) and try out some list commands:

```
lucy$ python
>>> MyList = ['a','b','c','d','e']
>>> MyList[:]
['a', 'b', 'c', 'd', 'e']
```

`MyList[:3]` returns the first three elements of the list, the same as `MyList[0:3]`:

```
>>> MyList[:3]
['a', 'b', 'c']
```

This can get confusing, since the first three elements have indices of 0, 1, and 2. The element with the index after the colon is not included in the result. This differs from the number before the colon. `MyList[2:]` returns all the elements from index 2 to the end of the list, the same as `MyList[2:5]`:

```
>>> MyList[2:]
['c', 'd', 'e']
```

The element with index 2, the `'c'`, is included in the result. Just imagine those tongs, and keep in mind that the index before the colon is inclusive, and the index after the colon is exclusive.

Using negative indices to count from the end of the list works the same as counting from the beginning of the list; it is merely that the reference point of the index has changed:

```
>>> MyList[-2:]
['d', 'e']
>>> MyList[:-2]
['a', 'b', 'c']
```

An additional colon followed by an integer can be added to the end of the range of indices. This last value, if specified, indicates the **step size** of the slice. By default, if it isn't specified, the step size is `1`, and all of the values in the range are returned. If a value larger than 1 is specified, then every *n*th element in the range will be returned:

```
>>> MyList[0:5:1] ← same as [0:5]
['a', 'b', 'c', 'd', 'e']
>>> MyList[0:5:2]
['a', 'c', 'e']
>>> MyList[::2]
['a', 'c', 'e']
```

A negative step size can also be specified, which reverses the order in which the elements are returned:

```
>>> MyList[::-1]
['e', 'd', 'c', 'b', 'a']
>>> MyList[::-2]
['e', 'c', 'a']
```

This allows you to quickly reverse your lists.

Unpacking more than one value from a list

Sometimes you want to create a new list when you are retrieving values from an existing list, but at other times the reason you are extracting particular elements from the list in the first place is because you want to put each in its own variable. It is simple to place the value of a single element at a time into a new variable, just by specifying its index, as with `i = x[0]`. You can also extract more than one variable at a time:

```
i,j = x[:2]
```

This unpacks the first two elements of the list `x` into the variables `i` and `j`. If `x` doesn't have at least two elements to begin with, this code generates an error. You can unpack any list in this way, as long as the number of elements you retrieve from the list matches the number of variables you are trying to put them into.

The `range()` function to define a list

The function `range()` takes parameters that are similar to those used to index lists, and generates a list of integers. The range starts at the first parameter and ends just before the last parameter:

```
>>> RangeList = range(0,6)
>>> RangeList
[0, 1, 2, 3, 4, 5]
```

The `range()` function works with negative numbers, but these behave differently than when referring to a subset of a list. For instance, `range(-5,6)` will generate a list of numbers from -5 to 5.

Use a third parameter to indicate a step size if you want the numbers in the list to be incremented by a value other than 1:

```
>>> range(1,10,2)
[1, 3, 5, 7, 9]
>>> range(10,0)   ← A lower limit of 10 doesn't work with 0 as the upper
[]
>>> range(0,10,-1) ← Likewise, stepping backward from 0 to 10 doesn't work
[]
>>> range(10,0,-1) ← Now you're talking
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> range(-5,-11,-1)
[-5, -6, -7, -8, -9, -10]
```

Notice that if the step size goes in the wrong direction, such as being negative when the end value is greater than the starting value, an empty list is returned. This is what happened in the second and third examples here.

The `range()` function simplifies a variety of tasks. Imagine that you want to create a vertical list of labels corresponding to the wells of a 96-well plate. By convention, the columns of these plates are designated with the numbers 1–12 and the rows with the letters A–H:

A1	A2	...	A12
B1	B2		B12
...			
H1	H2	...	H12

A quick way to do this is with a tiny program consisting of two **nested loops** that cycle through the eight letters and twelve numbers, printing all combinations. This program will take advantage of the `chr()` function, which returns the ASCII character based on its special number (see Chapter 1 and Appendix 6). In ASCII, capital `A` is `chr(65)`, `B` is `chr(66)`, and so on:[4]

```
#! /usr/bin/env python
for Let in range(65,73): ← Step through character number 65 to 72
  for Num in range(1,13): ← For each letter, step through numbers 1 to 12
    print chr(Let) + str(Num)
```

[4] The inverse function for `chr()`, to find out the ASCII number corresponding to a letter, is `ord()`, so `ord('A')` is `65` and `ord('a')` is `97`. These functions end up being useful in creating text that would be arduous to type.

The result of this miniature program is to print out a column of ninety-six labels which you could use to label the rows in a spreadsheet:

```
A1
A2
A3
...
H11
H12
```

In the first `for` loop, the variable `Let` steps through `65` to `72`, the integers corresponding to ASCII code for characters `A` through `H`.[5] The nested loop (that is, the loop within the loop) goes through twelve times for each of the values of `Let`, printing the combined letter and number each time through the loop. To print the labels in table format, add a comma to the end of the `print` line without any other characters after it. This tells the `print` command to suppress the end-of-line character that it usually adds. Also add another `print` statement by itself at the same indentation level as the `for Num` loop. This will print a line break each time the letter (equivalent to the row) increments:

```
#! /usr/bin/env python
for Let in range(65,73):
   for Num in range(1,13):
     print chr(Let) + str(Num),   ← The comma is important
   print   ← Prints a line end once for each letter, after 12 numbers have passed
```

The output of this modified script will be eight lines of twelve elements:

```
A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12
B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12
...
H1 H2 H3 H4 H5 H6 H7 H8 H9 H10 H11 H12
```

A comparison of lists and strings

Indexing can also be used with strings, which in some respects behave as if they were lists of characters. Both list and strings can be combined using the plus (+) operator, sorted, and iterated within a `for` loop. The most important difference between strings and lists is that individual elements of a string cannot be directly modified, as the elements of a list can be. See the box on the next page on copying and modifying Python variables for more details. Elements can be retrieved from a list and a string in the same way, but trying to change the value of a character in a string will result in an error, while modifying an element in a list is acceptable:

[5] For international characters, the corresponding function is `unichr()`, and fortunately, for the UTF-8 version of Unicode, the values corresponding to `A-Z` are the same as they are in ASCII.

```
lucy$ python
>>> SeqString = 'ACGTA'
>>> SeqList = ['A', 'C', 'G', 'T', 'A']
>>> SeqString[3]
'T'
>>> SeqList[3]
'T'
>>> SeqString[3]='U'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> SeqList[3]='U'
>>> SeqList
['A', 'C', 'G', 'U', 'A']
```

THE SUBTLETIES OF COPYING AND MODIFYING PYTHON VARIABLES In Python, copying a variable doesn't create a new variable with a new name and a new value; instead, it creates a new name that refers to the old value. After you copy a variable, both names point to the same place in computer memory. However, the value of most variable types in Python, including integers, floats, and strings, can't be changed. When you assign a new value to an existing variable, what really happens is that a whole new variable is created, with a different value but the same name. For example, when you create an integer `x=5` and set `y=x`, both names refer to the same 5. If you then assign `x` a new value, say `x=8`, a whole new variable with the value 8 and the name `x` is created, and the old `x` is deleted. The name `y` still points to the original 5. In the end, this means that you can change the value of `x` without worrying about what this will do to the value of `y`, and forget everything mentioned in this box.

On the other hand, the values of some Python variables, including lists and dictionaries, can be changed. When a new value is assigned to such a variable, no new variable is created and the old variable really has a new value. When you define `A` as the list `[1,2,3]`, and then define `B=A`, you have created a potentially confusing situation: if you change the contents of the list by altering `A`, the values of `B` change as well, because `B` is still pointing to this same now-modified collection.

To break such linkages when they are not wanted, you can copy the actual values of a list, rather than just its name. This is done by selecting the full range of the list (using a colon within brackets, without any specified beginning and end point), and then putting these into a new list. This copies the elements to a new list. In place of `B=A`, which just copies the list name, the statement for creating an actual copy of the list is `B=A[:]`. If the copy is created in this way, any modifications to `B` won't affect `A` and any modifications to `A` won't affect `B`.

Converting between lists and strings

There are several ways to convert between lists and strings. For example, using the `list` function it is simple to create a list of the characters present in a string. This function will try to convert any variable into a list format:


```
>>> MyString = 'abcdefg'
>>> MyList = list(MyString)
>>> MyList
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

A list of strings or characters can be joined together into a single string with the `.join()` method. This method can be a bit confusing, because it seems to operate backwards. Rather then being a list method that takes a string argument, it is a string method that takes a list argument. The string that it acts on is inserted between each of the elements of the list when they are stitched together:

```
>>> MyList = ['ab', 'cde', 'fghi']
>>> ''.join(MyList)
'abcdefghi'
>>> '\t'.join(MyList)
'ab\tcde\tfghi'
>>> ' '.join(MyList)
'ab cde fghi'
```

In the first example above, the string that `.join()` acts on is empty—that is, defined with an empty set of quotes—and so the strings in the list are joined together without any characters in between them. The `.join()` method is particularly useful for building up lines of tab-delimited text from lists of data, or for creating a single string from a list of characters.

Adding elements to lists

Earlier, you modified a list by setting one of the elements to a new value. You may be tempted to do something similar: *add* elements to a list by directly accessing their index within the list. However, given a 2-element list, defined by:

```
x = ['A','B']
```

you can't add a third element `x[2]='C'`. The assignment operator is used to change the value of existing list elements, but can't be used to create elements that don't already exist. You have to use the `.append()` function to build lists:

```
x.append('C')
```

You can't add element number 20 directly to a list without defining the intervening values. Nor can you begin a list using `.append()` on a new variable name, if that variable hasn't already been defined. You must instead create an empty list, with `x=[]`, and then build up a list by appending to that starting point.

To insert items between existing list elements, use two matching indices to specify the location of insertion:

```
>>> MyList=['a','e']   ← Define a list with two elements
>>> MyList[1:1] = ['b','c','d']   ← Insert into position 1
>>> MyList
['a','b','c','d','e']
```

Lists are convenient for organizing data that correspond to sequential integer values, for example, consecutive field site numbers from 0–19. This kind of sequence, though, isn't always useful for your application, since it might be missing some numbers: there might be twenty field sites, but they could be numbered 0–9 and 12–21 (maybe a storm took out sites 10–11 and you added a couple more). A list would not be ideal for organizing these data, because you can't add a 12th element without a 10th and 11th element. One solution would be to use a list and pad unused elements with a placeholder value, but then you need to be sure to account for these placeholders in later steps. Another solution would be to use a dictionary that has integer keys. Then the keys could be any value at all, as long as they are unique.

Removing elements from lists

To delete elements from a list, use the `del()` function, or else just reassign an empty list to those elements:

```
>>> MyList = range(10,20)
>>> MyList
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> MyList[2:5]=[]
>>> MyList
[10, 11, 15, 16, 17, 18, 19]
>>> MyList = range(10,20)
>>> MyList
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> del(MyList[2:5])
>>> MyList
[10, 11, 15, 16, 17, 18, 19]
```

Checking the contents of lists

Often it's important to know if a particular element is contained in a list, regardless of its exact position. The `in` operator makes this test and returns `True` or `False`:


```
>>> MyList = range(10,20)
>>> MyList
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> 11 in MyList
True
>>> 21 in MyList
False
```

Sorting lists

There are a couple of different ways to sort lists in Python. One is the list method `.sort()`, available since Python version 2.4:

```
>>> MyList = [4,3,6,5,2,9,0,8,1,7]
>>> MyList.sort()   ← You don't have to assign the output to a new variable
>>> MyList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Notice that `.sort()` doesn't return anything, not even a sorted list. This is because the list is sorted in place—in other words, the list that `.sort()` acts on is itself changed. Most Python variables can't be directly modified, but lists can be (see the previous box on copying and modifying Python variables).

At times you may want to get a sorted copy of your list, and leave the original list unchanged. In these cases, you will want to use the `sorted()` function with the original list as a parameter in `()`:

```
>>> MyList = [4,3,6,5,2,9,0,8,1,7]
>>> NewList=sorted(MyList)
>>> MyList   ← The original list is unchanged
[4, 3, 6, 5, 2, 9, 0, 8, 1, 7]
>>> NewList   ← The sorted list has been placed here
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Identifying unique elements in lists and strings

Often you aren't interested in all the elements in a list—you just want those which are unique. You may, for instance, want to know which character states are possible for a set of specimens. The `set()` function[6] can be used to summarize the unique elements in lists and the unique characters in strings. We won't work with the output of the `set()` function directly, but instead immediately convert it to a list:

[6] The `set()` function actually returns a special set-type variable, but we are skipping that distinction. The function became built-in to Python installations starting with version 2.4. To use it in version 2.3, add this command to the beginning of your program: `from sets import Set as set`.

```
>>> Colors = ['red','red','blue','green','blue']
>>> list(set(Colors))
['blue', 'green', 'red']
>>> DNASeq = 'ATG-TCTCATTCAAAG-CA'
>>> list(set(DNASeq))
['A', 'C', '-', 'T', 'G']
```

This approach to identifying unique elements can be applied to make much of the code you write more general. As an example, take the `dnacalc.py` program you wrote in Chapter 8. In its original incarnation, this program had dedicated code for each of the four nucleotides (`A`, `T`, `G`, and `C`). Earlier in this chapter you modified the program (the new version was called `compositioncalc1.py`) to loop over a list of the nucleotides, which was a much cleaner way to approach the problem, since you could use the same code to analyze each of the nucleotides in turn. Still, though, the nucleotides to be analyzed were hardcoded in the definition of `BaseList`. Now, with the `set()` function, you can extract the list of characters from the string itself. This program would give the same result as the previous version when applied to DNA, but could also be used if there are nonstandard nucleotides, or if you wanted to count the frequency of amino acids in proteins. In fact it is so general now that it can be used to calculate the frequency of characters in any string. This program is available in the scripts folder as `compositioncalc2.py`:

```
#!/usr/bin/env python
DNASeq = "ATGTCTCATTCAAAGCA"
SeqLength = float(len(DNASeq))

BaseList = list(set(DNASeq))
for Base in BaseList:
    Percent = 100 * DNASeq.count(Base) / SeqLength
    print "%s: %4.1f" % (Base,Percent)
```

List comprehension

Many analyses require batch modifications or calculations for each item in a list. For example, you might want to square each element in a list, or make a new list containing the length of each word in an existing list. You can't just say `WordList.upper()` or `NumberList*2` to transform each element individually. Transformations applied to a list aren't automatically applied to each item. Sometimes such an attempt will result in an error, while at other times it will modify the list as a whole rather than the elements. The `*` operator, for instance, creates a new list that consists of multiple copies of the original list placed end-to-end:


```
>>> MyList = range(0,5)
>>> MyList
[0, 1, 2, 3, 4]
>>> MyList * 2
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
```

There are modules, such as the `numpy` module described in Chapter 12, that do provide more sophistication for applying operations to elements in a list. A general solution would be to write a `for` loop that goes through each element in the list and applies the desired transformation:

```
>>> Values = range(1,11)
>>> Values
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> Squares = []
>>> for Value in Values:
...     Squares.append(Value**2)
...
>>> Squares
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

The code above[7] produces a list called `Squares` containing the squares of 1–10. (Note that indented code works fine at the Python interpreter: the prompt changes to `...` to let you know you are in a nested code block, and you can hit an extra return to get out of the code block.)

There is a shorthand construct called **list comprehension** that lets you perform this same kind of operation on each element in a list, but with a single command. This is a little bit complex to understand, but it can be a big time-saver in your programs. For example:

```
>>> Values = range(1,11)
>>> Values
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> Squares = [Element**2 for Element in Values]
>>> Squares
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

The calculations are made in a single line, rather than with a multiline loop. The list comprehension statement loops through the list `Values` and performs some operation (`**2` in this case) on each item (`Element`), and returns the list of results. Notice that the entire construct is within square brackets.

[7] The `**` operator is for doing exponents, so `x**y` is `x` to the `y` power.

List comprehension is a useful way to extract columns of data from a two-dimensional array or a list of strings. You can specify a single index to get a column, or a range to get a subset of each list element. For example:

```
>>> GeneList = ['ATTCAGAAT','TGTGAAAGT','TGTATCGCG','ATGTCTCTA']
>>> FirstCodons = [ Seq[0:3] for Seq in GeneList ]
>>> FirstCodons
['ATT', 'TGT', 'TGT', 'ATG']
```

In this case, the variable `Seq` takes on the value of each entire string in the list, and then the first three characters are extracted into a new list. Several operations can even be combined together in this stage. Here the first three characters of each string are extracted and concatenated to a string with the + operator:

```
>>> Linker='GAATTC'
>>> Start = [(Linker + Seq[0:3]) for Seq in GeneList ]
>>> Start
['GAATTCATT', 'GAATTCTGT', 'GAATTCTGT', 'GAATTCATG']
```

Here is another example of a function used inside a list comprehension, reminiscent of your first Python program. (The resulting list isn't stored in a variable, so the interactive prompt displays it directly):[8]

```
>>> [ Seq.count('A') for Seq in GeneList ]
[4, 3, 1, 2]
```

Although you can convert a string to a list of characters using the `list()` function, it is sometimes hard to go from a list of numbers `[1,2,3]` to the equivalent strings `['1','2','3']`. You can't just use `str(ListOfIntegers)`. List comprehension again comes to the rescue:

```
>>> [ str(N) for N in range(0,10) ]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

The format for list comprehension can be a bit confusing (try things out in the interactive prompt before putting them in your program), but if you work with matrices of data, you will probably find them to be very useful. If you need even better manipulation and access to the components of an array, see the section on the `numpy` and `matplotlib` modules in Chapter 12.

[8] You can even nest list comprehension loops. For example:
`[ [ Seq.count(Base) for Seq in GeneList] for Base in "ACGT" ]`

SUMMARY

You have learned how to:

- Start the Python command-line interpreter
- Use `dir()` to see functions within a variable
- Go to the Web for Python help
- Create logical expressions
- Write `if` statements
- Use the `else:` command
- Store data in a dictionary as `{key: value}` pairs
- Retrieve dictionary entries using `[]` or `.get()` after the dictionary name
- Convert data from the Web into programs by using regular expressions
- Write `for` loops to work with strings
- Use loops to look up values in dictionaries
- Work with lists as follows:
  - Define lists with `MyList =[1,2,3]`
  - Extract elements with `[]`
  - Add elements with `.append()`
  - Define numerical lists with `range()`
  - Convert strings to character lists with `list()`
  - Convert lists to strings with `' '.join()`
  - See if an item is in a list with the `in` function
  - Identify unique list elements with `list(set())`
  - Sort lists with `.sort()` and `sorted()`
  - Remove elements with `del()` or `[]`
- Use list comprehension, for example
  `Squares = [Val**2 for Val in MyList]`

Moving forward

- Indent the `print` statements in `proteincalc.py` so that they occur inside the `for` loop, and see how it affects the output of your program.
- Working at the interactive prompt, try constructing and sub-sampling multi-dimensional lists.
- Build a list-comprehension statement that returns a list of the characters `'a'` to `'z'`.

Chapter 10

READING AND WRITING FILES

The programs you have created so far rely on information being written into the program itself, or else provided by the user at a prompt. In most cases, though, you will want to process data that are stored in files. In this chapter you will learn how to open text files, parse the data, and then use that information to generate new text files. You will develop these skills by building a file converter program that reads in a tab-delimited file containing latitudes and longitudes, then writes out a file that can be visualized with **Google Earth**. In the course of building up this example, you will get experience in Python with file handling, regular expressions, several new functions, and a variety of other tools.

Surveying the goal

This chapter will focus on building up a program that can read location data from one text file format, convert it to another format, then rewrite it in another file. Specifically, this program will reformat an input file with a series of locations to create an output file that can be read and displayed by the geographical viewer **Google Earth**. This is a very typical challenge for biologists—data are in hand and you know what you want to do with them, but the format of the data isn't understood by the program you want to use.

The input file, `Marrus_claudanielis.txt`, contains the latitude, longitude, depth, and associated data for several specimens that were considered when a new species of siphonophore, *Marrus claudanielis*, was described. The first challenge is to simply read the text from the file. Next, the individual components of the data must be parsed—that is, extracted from each line. Finally, these data must be repackaged into a new format and written to an output file.