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.