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.

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

Use ctrl Z.

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.

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.

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 (`#!`).

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).

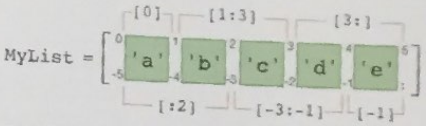

FIGURE 9.2 Numerical ways to think about indexing list elements

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 each element in a list of strings uppercase, 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]`