Chapter 5

HANDLING TEXT IN THE SHELL

Now that you are familiar with moving around in the shell, you can start to use it to do work. In this chapter you will begin to process large data files, viewing them, combining them, and extracting information from them. You will also see how to retrieve data from the Web at the command line.

Editing text files at the command line with `nano`

You are already familiar with editing text files through the graphical user interface of TextWrangler. It is sometimes more convenient, though, to create or modify a file right at the command line. Later, when you work on a remote machine, the only way to directly modify a file may be through a command-line editor.

Although `less`, which you learned about in the previous chapter, is a convenient command-line viewer for text files, it does not allow you to edit the contents of a file. For that, you will need one of the many command-line text editors available, such as `vi` or `emacs`. Each of these programs is optimized for different tasks and has its own devoted adherents. Each also has its own set of unique keystrokes for performing similar functions, so learning one of these editors doesn't necessarily enable you to easily use others. For starting out, we recommend `nano`, a widely available, general-purpose command-line text editor.[1] It displays the options in a menu-like array at the bottom of the screen.

If you call `nano` without any arguments it will create a new blank file. Move to your work folder and start a blank document:

```
host:~ lucy$ cd ~/pcfb/sandbox
host:pcfb lucy$ nano
```

[1] If your shell environment does not seem to have `nano`, try using `pico`.

Enter some text into the blank document:

```
Helpful shell commands
```

As you can see, most characters you type show up just as you would expect in the document. Along the bottom of the window, though, you will see a series of characters, each preceded by a caret (^), and each followed by a short description of a corresponding function (Figure 5.1). The caret signifies that to execute these functions, you hold down the ctrl key while pressing the relevant character.

Go ahead and try ctrl X, which exits nano. In this case, however, you won't actually exit. Instead, you will see the options at the bottom of the screen change. This is because the document has not been saved, and nano is wondering what you want to do with the contents. If the document had been saved (or had not been modified since it was opened), you would have returned immediately to the command line. As it is, above the new commands, you will see that nano is asking if you want to `Save modified buffer`. The options are `Y` for Yes, `N` for No, or `^C` (ctrl C) for Cancel. Just as you would expect from similar questions asked by programs with a graphical user interface, `Y` will save the file, `N` will discard any changes you have made, and `^C` will return you to editing the document.

Press `Y` to save the document, and nano will now give you a prompt that says `File Name to Write:` with a new set of options below. Type in `shelltips.txt`, and press return. The nano program will quit, and you will be back at the command line. Listing the directory contents with `ls` will show the `shelltips.txt` file you just created, and `less shelltips.txt` will let you view the file's contents. Open it again with nano, this time specifying the filename:

```
host:pcfb lucy$ nano shelltips.txt
```

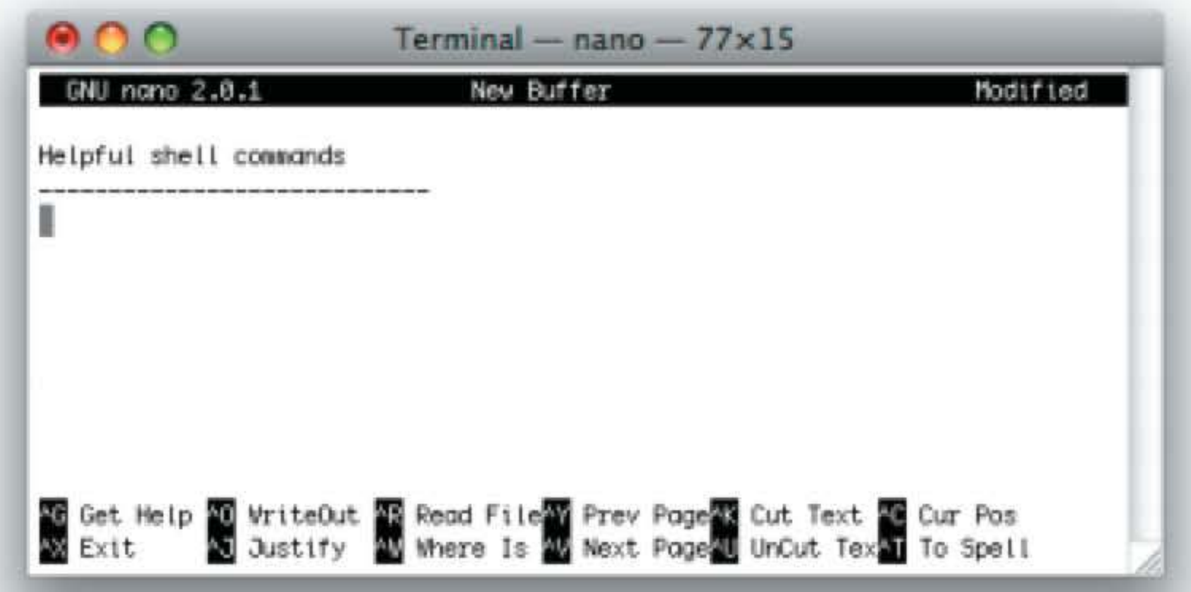

FIGURE 5.1 A view of the popular command-line text editor nano with the text you just entered Some of the available commands are displayed at the bottom of the window.

You can move the cursor with the arrow keys. In documents that don't fit in one screen, you can scroll up a page at a time with ctrl Y and down with ctrl V (again, you can see these listed among the key combinations at the bottom of the screen). Another useful key combination is ctrl O, which saves the file as you work without closing it. Once you are done exploring, exit nano with ctrl X.

You can delete the test file either the old-fashioned way in the GUI, by dragging it from the Finder window to the Trash, or from the command line

with `rm shelltips.txt`. Remember that `rm`, like the `rmdir` command for removing empty directories, deletes the file immediately after confirmation (depending on your configuration), without moving it to the Trash folder, so be careful when using it.

Although `nano` is available on all computers with OS X, you may find yourself working on a computer with a different type of Unix installation that doesn't include nano. In that case, see if `pico` is available. This is very similar to `nano`, and in fact `nano` is actually based on it. If `pico` isn't present on your computer either, you will probably need to give yourself a crash course on one of the other command-line text editors. You can also open a file in a separate TextWrangler window from the command line with the `edit` command, provided that you installed the TextWrangler command-line tools when prompted during installation. If you would like to install these tools after the fact, you can find the command to do so under the TextWrangler menu.

Controlling the flow of data in the shell

Redirecting output to a file with >

There are many times when it is useful to send the output of a program to a file rather than to the screen. Though this might at first sound like a nice but esoteric trick, it adds tremendous flexibility to the way that you can extract and combine data. It also creates the possibility of hooking software together in new ways. This is helpful when an analysis program normally sends its results to the screen, but you want to save the results instead. You could use the GUI's copy and paste functions to copy the results from the screen, but this can't be automated and gets cumbersome for big files or lots of files.

To redirect the output of a program to a file instead of the screen use >, the **right angle bracket** or greater-than sign found above the period on a U.S. keyboard. You type your command as you normally would, then on the same line add a > followed by the name of the file you want to send the output to. Think of > as an arrow that is pointing to where the output should go.

If the file doesn't already exist, the redirect will create it. Be careful, however: if a file with the same name does exist, it will be erased and replaced with the program output. It is therefore very easy to accidentally destroy an important document. (Later we will describe a way to avoid this behavior by redirecting with two right angle brackets together, that is, >>.)

To try out redirection, `cd` to the `sandbox` folder and use `ls` to list those files in the `examples` folder which end in `.seq`:

```
host:~/Desktop lucy$ cd ~/pcfb/sandbox/
host:sandbox lucy$ ls -l ../examples/*.seq
-rw-r--r--  1 lucy staff  524 Nov 1 2005 ../examples/FEC00001_1.seq
-rw-r--r--  1 lucy staff  600 Nov 1 2005 ../examples/FEC00002_1.seq
-rw-r--r--  1 lucy staff  538 Nov 1 2005 ../examples/FEC00003_1.seq
```

```
-rw-r--r--  1 lucy staff  622 Nov 1 2005 ../examples/FEC00004_1.seq
-rw-r--r--  1 lucy staff  490 Nov 1 2005 ../examples/FEC00005_1.seq
-rw-r--r--  1 lucy staff  548 Nov 1 2005 ../examples/FEC00005_2.seq
-rw-r--r--  1 lucy staff  495 Nov 1 2005 ../examples/FEC00006_1.seq
-rw-r--r--  1 lucy staff  455 Nov 1 2005 ../examples/FEC00007_1.seq
-rw-r--r--  1 lucy staff  501 Nov 1 2005 ../examples/FEC00007_2.seq
-rw-r--r--  1 lucy staff  569 Nov 1 2005 ../examples/FEC00007_3.seq
```

Now, try the same command, but redirect the output to a file called `files.txt` (you can just press ↑ and type the new text from > onward):

```
host:sandbox lucy$ ls -l ../examples/*.seq > files.txt
```

This time, there won't be any output to the screen and you'll get the command line right back. Check the contents of `sandbox` with `ls`, and you will see that the file `files.txt` has been created. Take a look at this new file with `less` or `nano`. It contains the exact text that `ls` would have sent to the screen if you had not redirected the output to a file. Also note that `files.txt` lists the contents of the `examples/FEC` folder, even though `files.txt` itself is in the `sandbox` directory. This is because the current working directory is `sandbox`, but by using `../` we told `ls` to look back starting in the `examples` folder instead.

It is not uncommon to want to create a file listing the contents of a directory, perhaps to send to a colleague, or to use as a starting point in automating a task involving those files. In these cases and others, a simple `ls` with a redirect can be very helpful.

Displaying and joining files with `cat`

Another useful command is `cat`. It is very simple, taking a list of one or more file names separated by spaces and outputting the contents of these files to the screen. Instead of displaying the contents in a special viewer or editor, as `less` and `nano` do, `cat` dumps them right into the display without a break, in the same manner as the results of many other commands, such as `ls`. Though this may not seem very useful at first, there is a good chance `cat` will become one of your most frequently used commands.

To begin with, `cat` is a convenient tool to view the contents of a short file. For example, `cat files.txt` will output the contents of the `files.txt` file from the previous example. You don't want to view large files with `cat`, as it will take some time for the file contents to scroll by. (Remember, `less` is very good at viewing large files because it only loads a small chunk of them into memory at a time.) If your file seems to be scrolling past for too long, press *ctrl* C to kill `cat` and get the command line back. This is a useful trick whenever a program stops responding in the terminal.

Take a look at the contents of another example file. From within sandbox, type the following command (or press tab after the F to partially complete the file name):

```
host:sandbox lucy$ cat ../examples/FEC00001_1.seq
```

As you might expect, this will dump the contents of the file FEC00001_1.seq in the examples directory to the screen.

Now, use cat to view two files. Notice that there is a space between the paths to the two files:

```
cat ../examples/FEC00001_1.seq ../examples/FEC00002_1.seq
```

You can see that cat just dumps the contents of both files to the screen, one after the other, without any separation of any kind between them. Now, let's look at the contents of all ten of the files at once that end in .seq. You could write out the path to each of them, but it is much easier to use a wildcard as we did for ls:

```
cat ../examples/*.seq
```

At this point, it is a simple matter to use cat in combination with a redirect to create a new file that contains all the contents of all the other files, joined end to end:

```
cat ../examples/*.seq > chaetognath.fasta
```

Combining files with GUI tools such as the Finder can be a tedious and frequently recurring task. This one-line command just saved you from opening ten different files, copying and pasting the contents of each file into a new blank document, and then saving the file you generated. Best of all, this command would have worked just as well for two files or for a hundred. It is an expandable solution that you only need to learn once for projects of any size. The other thing to note is that the command didn't just join all the files in a directory; it looked for only particular files that ended with .seq. This specificity was important since there are many other files in the examples directory that we didn't want to combine into chaetognath.fasta, and it saved you the added task of sorting through these files.

> When using wildcards with cat, do not use the same extension for the output file as the input file. If you try cat *.txt > combined.txt the shell may consider combined.txt to be one of the input files and attempt to continue writing it to itself ...forever. If you accidentally create a never-ending command, like this, remember that you can use ctrl C to stop it.

Notice that all the `.seq` file names have the general format `FEC00001_1.seq`, and that the digit after the underscore is either a `1`, `2`, or `3`. You might want to make a combined file from only the original files that have a `1` in this position. This could be done as follows:

```
cat ../examples/*1.seq > chaetognath_subset.fasta
```

The addition of the `1` after the `*` increased the specificity of the file names that are matched. If you wanted to now add the files with a `2` to `chaetognath_subset.fasta`, you could use a slightly modified redirect which appends to existing files:

```
cat ../examples/*2.seq >> chaetognath_subset.fasta
```

There are two things that are different about this command relative to the one preceding it. The `1` after the `*` was changed to a `2`, and another `>` was added so that the redirect is now `>>`. If you had just changed the `2` and rerun the query, `cat` would have found the right files and joined their content, but the `>` would have written over the existing `chaetognath_subset.fasta` file. It would no longer include the results from the first run. The `>>` behaves much as `>`, but it appends the redirected content to the end of a file if it already exists rather than replacing the file altogether. If the file doesn't already exist, it creates it just as `>` does, so in most cases you can default to using `>>` and avoid the risk of losing files by accident. You can remember the distinction by noting that the second `>` is added to the end of the first, just the like operation itself. Later you will learn about other redirects that send data from a file to a program, or from the output of one program directly to the input of another, without ever generating a file.

Regular expressions at the command line with `grep`

Working with a larger dataset

You are now going to dive into a larger dataset, a compilation of measurements taken by Gus Shaver and colleagues on plant harvests in the Arctic. It is available online[2] and is reproduced in the `examples` folder as a file called `shaver_etal.csv`. The file extension csv usually designates a file of comma-separated values, as it does in this case.

Open `shaver_etal.csv` with TextWrangler to get a sense of its overall layout. You can see that it is comma-delimited text. (We've only dealt with tab-delimited text so far, but any character can in theory be used to separate data fields in a text file.) The rows are different samples, and the columns are various

[2] Original source: `http://tinyurl.com/pcfb-toolik`. Also available at `practicalcomputing.org`.

measurements. The first row is a header that describes the measurements within each column. You'll also notice that many of the commas don't have data between them. Even when some data are missing (either because they weren't collected or weren't applicable to a particular sample), it is still important to have all the commas in place; this ensures that values coming after the missing data are interpreted as being within the correct column. The last rows don't have any data—they are all commas. (If you are really astute, you'll notice that there are two empty columns at the right side of the file as well.) This isn't uncommon in character-delimited files generated by spreadsheet programs, which will sometime write out empty rows or empty columns that once had data or are formatted differently.

Extracting particular rows from a file

What if you want a file that contains only those records from `shaver_etal.csv` which pertain to Toolik Lake? The command-line program `grep` is an easy-to-use tool that quickly extracts only those lines of a file that match a particular regular expression. This is the first time we will use the regular expression skills that you developed earlier—outside of the context of a GUI-based text editor such as TextWrangler. In this case, no wild cards or quantifiers are needed. The regular expression will be just the literal piece of text you are looking for, "Toolik Lake":

```
host:~/Desktop lucy$ cd ~/pcfb/sandbox/
host:sandbox lucy$ grep "Toolik Lake" ../examples/shaver_etal.csv
```

The first argument (`"Toolik Lake"`) is the regular expression, and the second argument specifies the source file you want it to examine. The `grep` program scans the file and displays only those lines that contain the search phrase. We needed to put quotes around our regular expression because it had a space in it; otherwise, `grep` would have considered `Toolik` to be the search term and `Lake` to be a separate argument.

In the previous example, the results were simply sent to the screen. Now use a redirect to send them to a file: press ↑ to recall your previous command and add `> toolik.csv` to the end:

```
grep "Toolik Lake" ../examples/shaver_etal.csv > toolik.csv
```

Now you have a file that is a subset of the original, containing only those lines with Toolik Lake. The only issue now is that the new file doesn't have a header. To solve this, you can copy the header and paste it in with nano or TextWrangler, or create another file that has just the header and then use `cat` to join it to the new file subset you made.

To tell `grep` to ignore the case of letters in search matches, so that you can find `toolik` as well as `TOOLIK` and `tOOlik`, add `-i` as an argument to the command.

> Sometimes when you type a command and hit return, the shell will appear to be frozen, showing only a blank line. This can happen if you made a mistake typing the command, such as forgetting to add a closing quote mark, and it usually just means that the shell is awaiting the rest of the command. Depending on what the status is, you can either try typing the remainder of the command and hitting return again, or using ctrl C to abort the command and start over.

Though you will often need to search a file for the lines that contain a known phrase, the example above didn't leverage the power of regular expressions at all; the search phrase was just a bit of literal text. The command-line version of `grep` uses slightly different syntax than the regular expressions available in TextWrangler.[3] For instance, `grep` doesn't understand `\d`, so you will need to specify the range `[0-9]` instead. The `man` file for `grep` (which you can see with the command `man grep`) explains some of the command-specific syntax, and you can consult it if something doesn't work as expected.

Now create a file that has only the lines from Toolik Lake that were recorded in August. One way to do this would be to take a two-step approach and just use `grep` to create a subset of the new file `toolik.csv`, composed of only those lines that have `Aug` in them. A more direct strategy would be to simultaneously search for lines that have both `Aug` and `Toolik Lake` in them. This requires a wild card and quantifier between the terms to accommodate the intervening text. You will again use the `.*` formulation from regular expressions, with `.` as the wildcard and `*` as the quantifier:

```
grep "Aug.*Toolik Lake" ../examples/shaver_etal.csv > toolik2.csv
```

Some complications arise with `grep` at the command line when searching for characters that have special meanings for both the shell and regular expressions. For these characters to be taken literally as part of the name and not as special characters, they need to be escaped out with a backslash, so that the shell passes them on to `grep`. For now, we'll leave it at that and let you explore this a bit more on your own if you like.

> When using regular expressions to search for subsequences or other data that might wrap across lines, if there is a line ending in the middle of the pattern it will not match. See the **agrep** command in Chapter 16 for ways to perform searches which span multiple lines.

You can cause `grep` to return all the lines that *don't* match your search expression by inserting `-v` in your command. Thus, if you had the opposite challenge and needed to construct a file with all the records *except* those from Toolik Lake, you could use this command:

```
host:sandbox lucy$ grep -v "Toolik Lake" ../examples/shaver_etal.csv
```

This time the header is included in the output since it doesn't contain `Toolik Lake`.

[3] See Appendix 2 for more information regarding differences in regular expressions syntax.

Redirecting output from one program to another with pipe |

You have already used > and >> to redirect the output of a program to a file. Using the **pipe** redirect, it is also possible to redirect the output of one command directly to the input of another, without ever generating a file. The pipe character is the vertical bar (|) located above the backslash key on your keyboard. In the previous `grep` example we specified a file to be examined. If you leave off the file name and only specify the regular expression, `grep` will take the input from another source. In this case, this other source will be the output of the pipe.

To illustrate this, we will use the command `history`, which displays all your most recent commands line by line. This is a great way to remember what you did, or to find a previous command so that you can reuse it. Try it out:

```
lucy$ history
```

It can sometimes be difficult to find a particular command in the hundreds of results that are displayed. One way to find the right one is if you can remember the argument values or other text you used in issuing that particular command. You can then use `history` to display your previous commands, and use `grep` to find those lines that contain the text of interest. Of course, you could redirect the output of history to a file with >, and then run `grep` on the new file; however, that gets cumbersome and generates a file that you will only need once, cluttering up your system. It is much more convenient to use a pipe to send the output of `history` directly to `grep`:

```
lucy$ history | grep Toolik
```

This will display all the commands you have executed that contain the word `Toolik`. The output of the `history` command does not depend on what directory you are in when you run it.

You can also use the pipe to construct searches by combining two consecutive `grep` operations. In the previous example regarding Toolik Lake, you created a regular expression that matched `"Aug.*Toolik"`. If you want to make one of these terms case-sensitive, but the other not, or if you want to invert one search and not the other, you can use a pipe. To do this, first run a `grep` command for lines containing `Aug`, and then instead of sending that output to the screen or a file, pipe to another `grep` command that looks for `Toolik`. This would give you more flexibility in constructing each particular search:

```
grep "Aug" ../examples/shaver_etal.csv | grep "Toolik" > toola2.csv
  ↳Original grep  ↳The searched file      ↳Piped to 2nd grep   ↳Redirected to file
```

Most but not all programs can accept data from a pipe in the way `grep` did above. It is not enough to just send the output of one program to another; the data must be organized in such a way that the receiving program can make sense of them. Since `grep` can handle any text, this isn't an issue in these particular examples.

Searching across multiple files with `grep`

To search the contents of multiple files in one step for a particular bit of text, you could use `cat` to join the contents of the files together and then feed them to `grep` with a pipe:

```
host:sandbox lucy$ cd ~/pcfb/examples/
host:examples lucy$ cat *.seq | grep ">"
>Fe_MM1_01A01
>Fe_MM1_01A02
>Fe_MM1_01A03
>Fe_MM1_01A04
>Fe_MM1_01A05
>Fe_MM1_01A06
>Fe_MM1_01A07
>Fe_MM1_01A08
>Fe_MM1_01A09
>Fe_MM1_01A10
```

Notice that the > in the command is within quotes. This indicates that > is the character that is being searched for, not a redirect to send the results to a file. This general approach is very useful for summarizing or extracting data spread across multiple files, but disastrous results can occur if you forget the quote marks.

In some cases, you may want to see both the lines with the pattern and the name of the file they were found in. This is the default behavior for `grep` when you designate input files with wildcards:

```
host:examples lucy$ grep ">" *.seq
FEC00001_1.seq:>Fe_MM1_01A01
FEC00002_1.seq:>Fe_MM1_01A02
FEC00003_1.seq:>Fe_MM1_01A03
FEC00004_1.seq:>Fe_MM1_01A04
FEC00005_1.seq:>Fe_MM1_01A05
FEC00005_2.seq:>Fe_MM1_01A06
FEC00006_1.seq:>Fe_MM1_01A07
FEC00007_1.seq:>Fe_MM1_01A08
FEC00007_2.seq:>Fe_MM1_01A09
FEC00007_3.seq:>Fe_MM1_01A10
```

Now you get the filename and contents of every matching line, separated by a colon.

Finally, it is also possible to list just the names of files that contain a particular text pattern. If you specify the `-l` argument to `grep`, instead of listing each line that matches, it outputs the name of each file that contains a matching line:

```
host:examples lucy$ grep -l "GAATTC" *.seq
FEC00001_1.seq
FEC00002_1.seq
FEC00004_1.seq
FEC00005_1.seq
FEC00005_2.seq
FEC00007_2.seq
```

Notice that the query pattern has been modified, and that only a subset of the files contain the specified pattern.

In conjunction with a redirect, the above command can generate a new file with these filenames:

```
host:examples lucy$ grep -l "GAATTC" *.seq > ../sandbox/has_EcoRI.txt
```

Refining the behavior of `grep`

There are a variety of other useful arguments that modify the behavior of `grep`; you can explore these in the `grep` manual page with `man grep`. Many of these are also listed in the table in Appendix 3, and some are listed here (Table 5.1). One of the most helpful is the argument `-c`, which will cause `grep` to output the number of lines which contain the specified pattern rather than the lines themselves. You could use it, for instance, to count the number of DNA sequences in a FASTA file: `grep -c ">"`

TABLE 5.1 Options that modify the behavior of `grep`

Usage example: `grep -ci text filename`	
`-c`	Show only a count of the results in the file
`-v`	Invert the search and show only lines that do *not* match
`-i`	Match without regard to case
`-E`	Use regular expression syntax (as described in Chapters 2 and 3) with the exception of wildcards; use `[ ]` to indicate character ranges, and enclose search terms in quotes. See Appendix 3.
`-l`	List only the file names containing matches
`-n`	Show the line numbers of the match
`-h`	Hide the filenames in the output

Be extremely careful if you are constructing a `grep` query that includes > as part of the search term. Be sure that it is within quotes, or else the shell will interpret it as a redirect and replace the contents of the file you wanted to search. When in doubt, use quotes. They are also necessary if the pattern (i.e., the query text) has wildcards or quantifiers.

A NOTE ABOUT `awk` AND `sed`

Two powerful commands called `awk` and `sed` are available in most shell environments. These are similar to `grep` in that they let you search and modify the contents of files, but they are like programming languages in that they have even more opportunities for performing complex tasks. Both `awk` and `sed` are tricky, and learning them does not necessarily tie in to other command-line skills. Because of this, and because of our focus on Python programming, we will not cover them here. However, if you are interested, you might read about them online and see if they are especially suited to your needs and inclinations.

will count the header lines, and therefore the number of sequences regardless of their length. It can also quickly ascertain the number of lines of data in a file. One way to do this is with `grep -c $`, which returns the number of line endings and therefore the number of lines. (The command `wc file.txt` will give you a count of the characters, words, and lines in the file.) Remember that `-c` is showing the number of lines that contain the pattern, not the number of times the pattern matches, so if some lines have multiple matches, `-c` will underestimate their total number.

When searching through a long file it can be helpful to see not only the lines that contain your pattern of interest, but also where the lines are in the file. By default, `grep` will output the lines in the order they are encountered, but `-n` adds further location information by prepending the line numbers to the output.

These arguments can be used in combination with the others mentioned above: `-i` for case insensitive and `-v` for inverted searches. The `-v` option is important for some tasks. Instead of trying to figure out the regular expression to search for "lines that don't contain >" (Hmm... `"^[^>]*$"?`) you can just run a `grep` command for `">"` with the `-v` option. (Remember to use quote marks around >.) You also don't have to think about how to make a `grep` search that matches lines with three different items—just chain together three independent searches with the pipe, and the final output will be a consensus.

Retrieving Web content using `curl`

In the course of many projects, it is necessary to fetch data from an online database or other Internet resource. The standard way to do this is to browse to a Web page and then either click to download a linked file, or copy and paste text from the browser window into another document. These methods become impractical when data are spread across multiple pages, or when the data need to be updated from the source on a regular basis. Fortunately, a shell program called `curl` (that is, "see URL"), can directly access Internet files on your behalf, downloading the text content of a Web page without any need of a browser window. Later you will learn how to integrate `curl` with other tools to create automated workflows that can draw on outside data.

See Appendix 1 for alternatives.

The simplest `curl` command consists of `curl` followed by the Web address (also called a URL) of the item you want to retrieve. The URL is the line that shows up in the address bar of a web browser window, usually beginning with `http://` (Figure 5.2). Because this http text is so common as to almost go without saying,

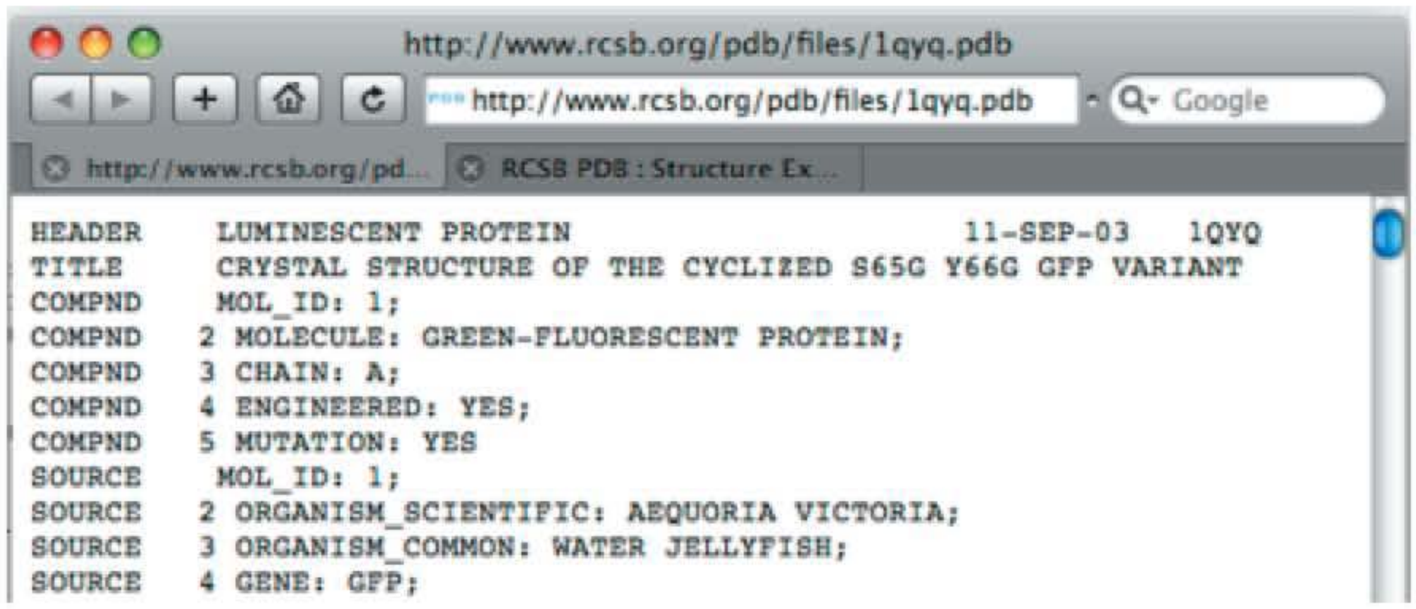

FIGURE 5.2 Example of a URL displayed in the address bar of a web browser

we omit it from many of the URLs in the book, but you should include it in your `curl` commands.

In the shell window, type the following command (or copy and paste it from `~/pcfb/scripts/shellscripts.sh`):

```
curl "http://www.rcsb.org/pdb/files/1ema.pdb"
```

When you press return, the content of that address (a file describing 3-D protein structure) will be downloaded to your terminal so that it scrolls across the screen. If you want to save the information to a file instead, you can use the redirect > followed by a file name:

```
curl "http://www.rcsb.org/pdb/files/1ema.pdb" > 1ema.pdb
```

To store files without using the redirect operator, use the `-o` option (output) followed by the destination file name. In this case you can say:

```
curl "http://www.rcsb.org/pdb/files/1ema.pdb" -o 1ema.pdb
```

The `curl` command really begins to become powerful when you use it to download a range of files in one step. For example, to retrieve weather data for each day of a month, specify the range of days in brackets `[01-30]`. This command will retrieve each day's record from August:

```
curl "http://www.wunderground.com/history/airport/MIA/1992/08/[01-30]/
DailyHistory.html?&format=1" >> miamiweather.txt
```

The command should be entered all on one line. (Again, you can also copy and paste it from the `shellscripts.sh` example file in your `~/pcfb/scripts/`

folder.) It generates thirty separate URLs in sequence, and downloads them one after the other. Because of this, you will need to use the append redirect >> instead of just >, or else each data file will overwrite the previously downloaded file.

An alternative to >> is to save the files using `curl`'s `-o` option again. By default, `curl` will only save the last page to that file name, rather than store all the results from the same command end-to-end; but as we shall soon see, this behavior for `-o` is easily modified to save to multiple files, each bearing a unique name corresponding to the appropriate value in the range you specified. The current value being retrieved within the range is stored in memory as a variable called `#1` (reminiscent of the `\1` used in regular expression searches). This can be used to generate unique file names for each of the individual results retrieved.

So to download weather data from the first day of each month and save each day as a unique file, use the following command, where the month field is represented by a range instead of the day:

```
curl "http://www.wunderground.com/history/airport/MIA/1979/[01-12]/01/
DailyHistory.html?&format=1" -o Miami_1979_#1.txt
```

Twelve files are retrieved and saved, with the `#1` in the file name substituted with `01`, `02`, on up to `12`. Take a look at the names of the saved files:

```
host:sandbox lucy$ ls Mi*
Miami_1979_01.txt   Miami_1979_04.txt   Miami_1979_07.txt   Miami_1979_10.txt
Miami_1979_02.txt   Miami_1979_05.txt   Miami_1979_08.txt   Miami_1979_11.txt
Miami_1979_03.txt   Miami_1979_06.txt   Miami_1979_09.txt   Miami_1979_12.txt
```

Ranges can also use letters `[a-z]`, and they are smart enough to add padding characters, depending on how you write them. So `[001-100]` will generate URLs in the form `file001` rather than `file1`, with the extra zeros as padding. You can also retrieve data from two ranges simultaneously (e.g., if numbered files exist within numbered directories); in that case, `#2` would be the placeholder for inserting the second range of values into a file name.

Rather than provide `curl` with a sequential range of file names, you may wish to retrieve from a list of URLs where only one portion of the address is changing at a time, but not in a predictable manner. To form this kind of query,

> Be sure you **cd** into your **~/pcfb/sandbox** or an appropriate destination folder before running these commands, because they can quickly generate large numbers of files that will clutter your home directory. If the command gets out of hand, perhaps taking longer or generating more files than you expected, you can interrupt with ctrl C at any time.

put the list of elements in curly brackets {} separated by commas. For instance, to retrieve a set of four particular protein structures at once, you could use:

```
curl "http://www.rcsb.org/pdb/files/{1ema,1gfl,1g7k,1xmz}.pdb"
```

There are many variations on the queries that you can form with `curl`. Take a look at the `man` page for `curl` for more information on how to tailor your Web downloads.

Other shell commands

The shell commands we have introduced so far will enable you to perform a wide range of tasks, but we have only barely scratched the surface of what is available. In Chapters 16, 20, and Appendix 3 you will encounter a variety of other commands that are very helpful for handling and analyzing data; these include `sort`, `uniq`, and `cut`. With these commands you will be able to do things like count the number of unique entries in a data file—even if it is gigabytes in size, a common challenge that applies to many dataset types. If you find that the command-line skills you have picked up so far are having a big impact on your data analysis abilities, you may want to briefly peek ahead to Chapter 16 and Appendix 3, either now or as you continue with the next chapter; these supplement the material presented here, and provide tips for jumping-off points to learn still more shell commands.

SUMMARY

You have learned how to:

- Edit files at the command line with `nano`
- Send output to a file instead of the screen with > and >>
- Join multiple files together into one file with `cat`
- Use `grep` to extract particular lines from a file
- Pipe output to another command with |
- Download data from the Web at the command line with `curl`