Chapter 4

COMMAND-LINE OPERATIONS: THE SHELL

In this chapter you will learn the rudiments of the shell, a tool for interacting with your computer through typed instructions at the command line. For now the focus will be on navigating your computer from the command line, with later chapters focusing on handling text, using programs, and automating tasks with this powerful interface.

Getting started: Don't fear the command line

In order to appreciate what the shell is, it is useful to contrast it with something more familiar to you: a **graphical user interface**. Graphical user interfaces, or GUIs, are the displays through which most users interact with their computer. Some examples of GUIs include the interfaces for the Windows and Mac OS X operating systems, the KDE and GNOME interfaces for Linux, and X Window. With each of these, you can open and manage files and folders, as well as interact with programs in an intuitive way, using a mouse to move a cursor among icons and menus on the screen.

GUIs have dramatically improved the friendliness and usability of computers for many tasks. These advantages do come with trade-offs, though, some of which are felt far more acutely by scientists than by other users. Potential disadvantages of GUIs include the following:

- Many of the analyses that scientists need to perform consist of a long sequence of operations which may need to be repeated on different datasets, or with slight modifications on the same dataset. This does not lend itself well to GUIs. Everyone has had the frustrating experience of clicking through the same sequence of menus and dialog boxes over and over again.

- Most programs do not keep a log of all the user commands issued through the GUI, and if you want to recreate the steps it took you to analyze your data, they must be documented separately.
- GUIs are not conducive to controlling analyses on a cluster of computers or on a remote machine. This will be a problem when you need to access increased computing power beyond your personal computer for complex or large-scale tasks.
- GUIs are labor-intensive to make and typically work on only the particular operating system for which they were developed.

Fortunately, there is another way of interacting with your computer that avoids many of these problems. This is the oft-dreaded **command line**, an entirely text-based interface. Using the command line requires an up-front investment in a new skill set, but provides a huge net gain in the long run for many data management and analysis tasks. To the uninitiated, the command line can seem like a primitive throwback to the days before the mouse. However, the command line is alive and well, and is in fact the interface of choice for a wide range of computational tools. In many cases it is a more convenient environment for data manipulation and analysis than a GUI could ever be.

We will start by explaining what the command line is and how to navigate your computer from this powerful interface. Later chapters will illustrate what you can do with further commands, particularly for text manipulation. This book is intended to get you comfortable with the command-line environment, but it is not a comprehensive guide. To go further, you can consult Appendix 3 on shell commands, as well as online references and any of the many books devoted to the topic. (We recommend a few in particular at the end of this chapter.)

Starting the shell and getting oriented

Starting the shell

Windows users, see Appendix 1; Linux users can use their Terminal program.

Shells run in what are called terminal emulators, or more simply, terminals. In Mac OS X, the default terminal program is in fact called Terminal, and it is located in the Utilities folder within the Applications folder.[1] Launch the program by double-clicking its icon. The terminal window opens with a greeting, a prompt, and a cursor (Figure 4.1). The greeting is usually brief, indicating information such as when you last logged in. The exact text of the prompt will vary, and can depend upon your computer's name, your user name, your current network connection, and other factors. Throughout this book we are going to assume you are a computer guru with the user name `lucy`, so the text of the demonstration prompts will include the word "lucy."

[1] Another terminal program sometimes installed on OS X is `xterm`, available through a menu for the `X11` application. Since `xterm` operates differently than `Terminal`, avoid using it for these sections of the book.

FIGURE 4.1 A portion of the Terminal window in OS X, showing the greeting, prompt, and cursor

The command line, as the prompt and cursor are known, is displayed within the terminal window by a program called a **shell**. Shells are customizable and powerful—and at times they can seem frustratingly simple-minded. You can issue short commands or write simple scripts that do amazing things, but you can also wipe out a chunk of your hard disk with a few ill-chosen punctuation marks. The shell assumes that you mean what you say, and rarely gives you the chance to opt out of a command you issue or to undo the results.

There are many different shell programs, and each has its own camp of loyal advocates. Shells mostly have the same capabilities, but employ subtly different ways of saying things. OS X has several shells installed that can be used in the Terminal window, but since version 10.3 the default has been the `bash` shell. We will use `bash` throughout this book.

It's time to try your first command. To check that you are set up to use the `bash` shell, at the prompt in the open terminal window type the following:

```
echo $SHELL  ← This name must be in uppercase
```

The terminal will respond by printing out the name of the active shell, which should be `/bin/bash`. You can also look at the top of the Terminal window to see if it says `bash`. If the active shell is not `bash`, open Terminal preferences, change

COMMAND-LINE INTERFACES ON DIFFERENT OPERATING SYSTEMS The examples in this book assume you are using the OS X operating system provided with Apple computers, which is a Unix operating system. If you are using an operating system other than OS X, see Appendix 1 for specifics about your system. Many other operating systems that are also based on Unix, such as Linux, are similar enough that most of the examples will work as written or with very minor modifications. In fact, the exact same command-line programs are widely distributed on many Unix systems. However, if you are on a Windows system, *you cannot follow along at the DOS or PowerShell prompt.* The Microsoft command-line interface provided with Windows operating systems is fundamentally different than, and not equivalent to, the **Unix** command line. As described in Appendix 1, you can install a Unix-like command-line tool called Cygwin in Windows which provides partial functionality. You can also install Linux alongside Windows to take full advantage of these command-line skills.

the default startup shell to `/bin/bash`, and relaunch Terminal by quitting and reopening it.[2]

A command-line view of the filesystem

The nested hierarchy of folders and files on your computer is called the **filesystem**. You should already be familiar with the filesystem through the variety of windows you see when using the graphical user interface of your computer. For example, you have browsed to folders and files within the Save and Open dialog boxes of various programs, and on OS X, you have used the Finder to create new folders and drag files between folders. These graphical dialog boxes and tools give you a bird's-eye view of how files are organized on your computer.

Your perspective of the filesystem from the command line is different than the detached view provided by the GUI. It is more like a first-person perspective—as if at any given time, you are viewing the filesystem from where you are standing within it. You get a different view depending on where you stand. Sometimes it is easier to stay right where you are and reach out to interact with a file from afar; other times, it is more convenient to first move to where the file is, and then work with it at close range.

When working from a graphical user interface such as the Finder, it is often unclear what is meant by the **root directory**—that is, the most inclusive folder on the system, serving as the container for all other files and folders. (Directory and folder mean the same thing; however, it is more common when working with the command line to speak of directories than of folders.) As you will see, the meaning of the root directory is much more obvious, as well as more important to know, when working at the command line. A Unix-based system such as OS X has a single root directory. You can think of it as the base of the tree of files on your computer, or as the most inclusive container that contains all other folders and files (Figure 4.2). Even the hard drive is located within this single filesystem sprouting from root. If you insert a disk into your DVD drive or plug in a thumb drive, these also get their own folders within the tree.

Misunderstandings of just what constitutes the root directory in Unix are common. Windows users are more used to each disk drive having its own root, in effect creating multiple roots (`C:`, `D:`, etc.) depending on the hardware configuration of the computer. On Mac OS X, some users think of their Home folder as the root directory, but in fact each user of that computer has their own Home folder; it is merely the default location for personal account settings and user-generated files. Other users get the impression that the root of the filesystem is their Desktop, but in fact the Desktop is just a folder nested within the Home folder.

When working with the command line, your view of the filesystem is always based on the directory you are currently in. So the first thing you need to learn

[2] If you cannot find an appropriate setting, join us in the PCfB support forum at `practicalcomputing.org`.

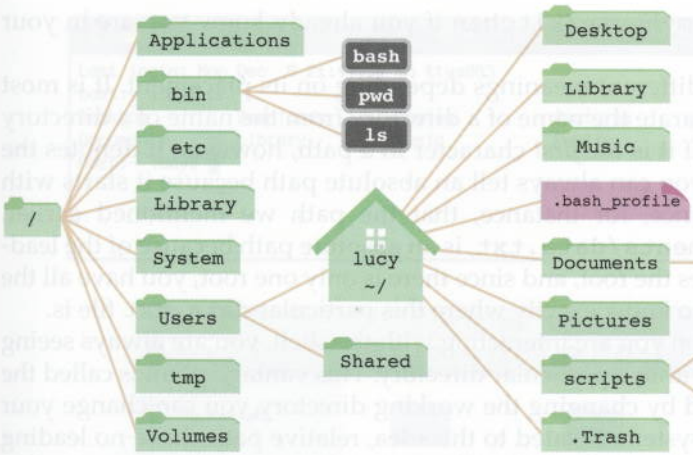

FIGURE 4.2 A portion of the filesystem structure in tree format Folders (also called directories) are in green, files (also called documents) are in maroon, and Unix programs (also called commands) are in gray. The root directory is at the far left.

is how to move to different locations, figure out where you are, and look around from there to see what other files and directories are close by.

The path

The written description of where something is located in the filesystem is called the **path**. The path is a list of directory names separated by slashes. If the path is for a file, then the last element of the path is the file's name. In Unix-based operating systems, the separator is a normal forward slash (/), *not* a backslash (\) as in Microsoft DOS. An example of a path is `/Users/lucy/Documents/data.txt`, which points to the file called `data.txt` in the Documents folder in the lucy folder in the Users folder in the root directory. Paths can refer to files or to folders. For instance, in this example, `/Users/lucy/Documents/` indicates the folder that contains `data.txt`, or in other words, the Documents folder for the user lucy.

A path can be absolute or relative. The **absolute path** is a complete and unambiguous description of where something is in relation to the root, the very base of the filesystem. Since there is only one root in the filesystem, you don't need any other information to know exactly which file or folder is designated by an absolute path. In some ways this is like knowing the latitude and longitude of a file or folder, in that it describes an unambiguous location. In contrast, a **relative path** describes where a folder or file is in relation to another folder.

In order to interpret a relative path you need to know the reference point as well. A relative path is essentially an offset. In this way it is like saying "the kitchen in this house." In order to know where the kitchen is, you need to know where the house is. Why even bother with relative paths when absolute paths can always be used unambiguously? In large part this is a matter of convenience. It just gets too cumbersome to fully specify the location of each object, especially as you move further and further away from the root. For example, why say

/mycity/mystreet/myhouse/kitchen if you already know you are in your house?

The slash has two different meanings depending on its placement. It is most commonly used to separate the name of a directory from the name of a directory or file which follows. If it is the *first* character in a path, however, it denotes the root directory. In fact you can always tell an absolute path because it starts with /. It is clear at a glance, for instance, that the path we mentioned earlier, /Users/lucy/Documents/data.txt, is an absolute path because of the leading slash that designates the root, and since there is only one root, you have all the information you need to know exactly where this particular data.txt file is.

Your home path will differ.

Remember that when you are interacting with the shell, you are always seeing your filesystem from within a particular directory. This vantage point is called the **working directory**, and by changing the working directory you can change your perspective on the filesystem. Related to this idea, relative paths have no leading slash and start with the first letter of the file or directory name. They describe the path to a file or folder in relation to the current working directory.

If the working directory were /Users/lucy/, for instance, then the relative path Documents/data.txt would specify the same file as the absolute path /Users/lucy/Documents/data.txt. Notice there is no leading slash in Documents/data.txt, so you know that this isn't an absolute path, and it is interpreted as relative to where you are. (In practice, it usually doesn't matter whether or not there is a trailing slash at the end of a path to a folder.) If you refer to /Documents/data.txt, you will get an error because of the leading slash; there probably is no folder and file by that name relative to the root of your system.

The relative path to a file in your working directory is just the name of the file itself, since it is in the same folder as you are. For example, within the folder /Users/lucy/Documents/, the relative path data.txt by itself is sufficient to specify /Users/lucy/Documents/data.txt. In practice, this is how you'll usually end up interacting with the filesystem from the command line. That is, if you want to perform a set of operations on a file, you'll first change your working directory to the folder that contains the file and then do everything from close range.

Navigating your computer from the shell

Listing files with ls *and figuring out where you are with* pwd

Each time you open a new terminal window, you are logging onto the Unix system of your computer anew. You will be located in what is known in Unix as your home directory, just as if you had clicked on the house icon representing your home folder in a Finder window. Each user with an account on your computer has their own home directory, in the form of their username, and all of these directories are located within the directory /Users. Since we assume the username lucy throughout the book (your username may of course be different), the home folder we'll use in all examples is /Users/lucy.

/mycity/mystreet/myhouse/kitchen if you already know you are in your house?

The slash has two different meanings depending on its placement. It is most commonly used to separate the name of a directory from the name of a directory or file which follows. If it is the *first* character in a path, however, it denotes the root directory. In fact you can always tell an absolute path because it starts with /. It is clear at a glance, for instance, that the path we mentioned earlier, /Users/lucy/Documents/data.txt, is an absolute path because of the leading slash that designates the root, and since there is only one root, you have all the information you need to know exactly where this particular data.txt file is.

Your home path will differ.

Remember that when you are interacting with the shell, you are always seeing your filesystem from within a particular directory. This vantage point is called the **working directory**, and by changing the working directory you can change your perspective on the filesystem. Related to this idea, relative paths have no leading slash and start with the first letter of the file or directory name. They describe the path to a file or folder in relation to the current working directory.

If the working directory were /Users/lucy/, for instance, then the relative path Documents/data.txt would specify the same file as the absolute path /Users/lucy/Documents/data.txt. Notice there is no leading slash in Documents/data.txt, so you know that this isn't an absolute path, and it is interpreted as relative to where you are. (In practice, it usually doesn't matter whether or not there is a trailing slash at the end of a path to a folder.) If you refer to /Documents/data.txt, you will get an error because of the leading slash; there probably is no folder and file by that name relative to the root of your system.

The relative path to a file in your working directory is just the name of the file itself, since it is in the same folder as you are. For example, within the folder /Users/lucy/Documents/, the relative path data.txt by itself is sufficient to specify /Users/lucy/Documents/data.txt. In practice, this is how you'll usually end up interacting with the filesystem from the command line. That is, if you want to perform a set of operations on a file, you'll first change your working directory to the folder that contains the file and then do everything from close range.

Navigating your computer from the shell

Listing files with ls *and figuring out where you are with* pwd

Each time you open a new terminal window, you are logging onto the Unix system of your computer anew. You will be located in what is known in Unix as your home directory, just as if you had clicked on the house icon representing your home folder in a Finder window. Each user with an account on your computer has their own home directory, in the form of their username, and all of these directories are located within the directory /Users. Since we assume the username lucy throughout the book (your username may of course be different), the home folder we'll use in all examples is /Users/lucy.

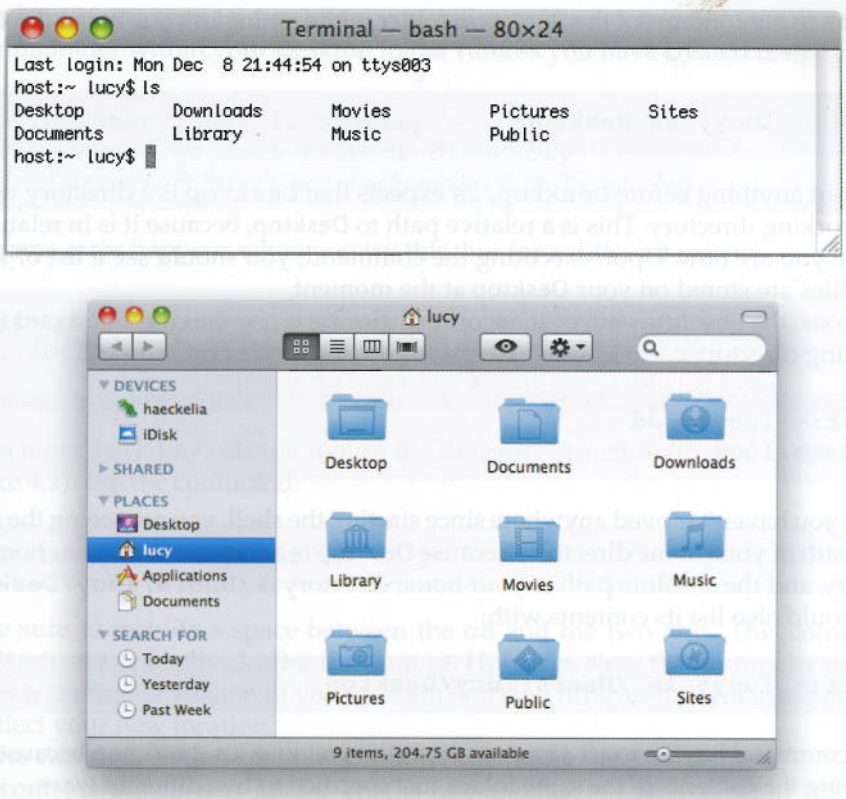

FIGURE 4.3 Viewing the interior of the home folder from two different interfaces, the Terminal window (top) and the familiar Finder window (bottom) Notice that the same folders are listed by both interfaces.

The first command you will learn, and one you'll use frequently, is `ls` (short for list). It prints out a list of files and directories contained within a specified directory. If you don't specify a directory, it shows you what is in the working directory—that is, the directory where you are currently located. Since all you have done so far is open a terminal window and start a new shell, by default the working directory is your home directory (`/Users/lucy`). Thus, `ls` shows you its contents (Figure 4.3).

In these examples, the bold characters are what you type, and the regular characters are what the system prints, and we may or may not show the prompt that begins each line. In place of `lucy` you'll see your own user name, and you will also see a different host name at the beginning of the line. This is your network identity, created when you first set up an account on your computer.

Type `ls` by itself, then use another folder name from that list.

To see the contents of the `Desktop` folder while sitting in your home directory, type:

```
host:~ lucy$ ls Desktop
```

On other systems you may not have a Desktop folder.

Without anything before `Desktop`, `ls` expects that `Desktop` is a directory within the working directory. This is a relative path to `Desktop`, because it is in relation to where you are now. Upon executing the command, you should see a list of whatever files are stored on your `Desktop` at the moment.

To see the absolute path of the working directory, use the command `pwd` (**p**rint **w**orking **d**irectory):

```
host:~ lucy$ pwd
/Users/lucy
```

Since you haven't moved anywhere since starting the shell, you are seeing the absolute path of your home directory. Because `Desktop` is a folder within your home directory, and the absolute path to your home directory is `/Users/lucy/Desktop`, you could also list its contents with:

```
host:~ lucy$ ls /Users/lucy/Desktop
```

This command has the exact same results as the previous `ls Desktop` because it is showing the contents of the same folder, just specified in two different ways.

How to move around with `cd`

To move to another directory, use the command `cd` (**c**hange **d**irectory). Move inside your `Desktop` directory by typing

```
host:~ lucy$ cd Desktop
host:Desktop lucy$
```

Notice that the prompt changes from `host:~ lucy$` to `host:Desktop lucy$`, showing the name of the new working directory.

You will need to keep in mind that capitalization generally counts in all types of Unix, including OS X. Getting the capitalization of a command wrong can be as bad as misspelling it. This capitalization behavior is hard to predict, though, so you should not rely on it to distinguish similarly named files.

To see the contents of the `Desktop` directory, but this time from within the directory, you can now type `ls` by itself:

```
host:Desktop lucy$ ls
```

From here, if you type `ls Desktop`, you will generate an error, because there is no Desktop folder within your Desktop folder (unless you have created one).

```
host:Desktop lucy$ ls Desktop
ls: Desktop: No such file or directory
    ↳Gives an error because you already moved to the Desktop folder
```

The same error happens when you try this, but for a different reason:

```
host:Desktop lucy$ ls /Desktop
ls: /Desktop: No such file or directory
     ↳Gives an error because the root folder doesn't have a Desktop folder
```

To move back towards the root in the directory structure (that is, to the left in Figure 4.2), use the command:

```
host:Desktop lucy$ cd ..
```

Make sure to include a space between the cd and the two dots. This command won't return any feedback after the prompt. However, since the prompt by default shows a shortened version of your current working directory, it will again change to reflect your new location.

The two dots in the command indicate the folder that contains the current folder. This is just another type of relative path. This will always move you from the current working directory to the directory which contains it—unless, of course, you are in the root directory, which is absolute and contained by nothing.

The `..` symbol can be used in conjunction with other directory names. This allows you to move with a single command from one directory to another that is sister to it in the hierarchy. For example, if you want to go from your Desktop directory to your Documents folder, this is equivalent to backing out one level (left in Figure 4.2) and then moving one level into a different but parallel directory relative to where you started. From the Desktop folder you would type:

```
host:Desktop lucy$ cd ../Documents
```

From `Documents` you can type `cd ..` to go back to the home directory again. Note that you can use `..` more than once in a command. For example, `cd ../..` will send you back two directories in one step.

A convenient way to enter long paths in the terminal window is to just drag and drop the file or folder from the **Finder** into a terminal window. So if you are working in a folder in the GUI or editing a file in another program, you can type **cd** and a space, then drop the folder icon to fill in the rest.

Signifying the home directory with ~

A shortcut for referring to your home directory is the tilde symbol (~, the wavy line below the esc key on most keyboards). So, for instance, no matter where you are on the system, you can type:

```
host:Desktop lucy$ cd ~/Documents
```

to jump straight to the Documents folder in your home directory. This is equivalent to typing the absolute path, specified all the way from root:

```
host:Desktop lucy$ cd /Users/lucy/Documents
```

Think of the text that follows ~/ as another relative path. However, instead of being relative to the working directory where you are now, it is relative to the home directory of whoever you are logged in as. If you are lost in your filesystem, you can still refer easily to files in your home directory using ~/ at the beginning of a path, just as you can take yourself directly home with `cd ~` . (The `cd` command by itself will also take you to your home directory.) Don't hesitate to use `pwd` to figure out what directory you are in at any given point.

If you are in the middle of a command and wish to start over, typing ctrl C will clear the line. (Hold down the key labeled ctrl or control while pressing C.) This key sequence is a universal way to interrupt a program or process.[3]

Adding and removing directories with `mkdir` *and* `rmdir`

So far we've discussed moving around the filesystem, but you have been a fly on the wall and have not actually done anything to affect the filesystem in your travels. One basic modification you can make to your filesystem is to add a directory—a task accomplished with the command `mkdir` (abbreviated from **make dir**ectory).

The following sequence of commands will make a folder called latlon in the sandbox folder you installed along with the example folder in Chapter 1. While you do these operations, it is illustrative to have a graphical browser window open in the Finder with the contents of the `pcfb/sandbox` folder showing. The `ls` commands give a before and after view of the changes resulting from `mkdir`:

```
host:~ lucy$ cd ~/pcfb/sandbox ←Only there if you installed examples
host:sandbox lucy$ ls
host:sandbox lucy$ mkdir latlon
host:sandbox lucy$ ls
latlon
```

[3] Unfortunately, Windows uses this command for copying, so it is not quite as universal as it might be. If you are working with Cygwin, as described in Appendix 1, you might have to re-learn some keyboard shortcuts.

```
host:sandbox lucy$ cd latlon
host:latlon lucy$ ls
host:latlon lucy$ cd ..
host:sandbox lucy$ ls
latlon
```

After the `mkdir` command is issued, you find a new directory has been created called `latlon`. If you look at your `sandbox` folder using the Finder (remember that GUI you used to use?) you should also see it there, represented by a new folder of the same name. In the last few commands you simply moved into the new directory, looked around (there was nothing there to see since you hadn't put any files in it), and moved back out. Next you get rid of the empty directory with the command `rmdir` (**r**emove **dir**ectory):

```
host:sandbox lucy$ rmdir latlon
host:sandbox lucy$ ls
```

You can see that `latlon` is now gone. By default, `rmdir` is conservative in that it only removes empty directories. The equivalent to `rmdir` for deleting files is `rm`. Use caution with `rmdir` and even more so with `rm`. These commands are not like putting something in the Trash, where you can pull it out later. Files deleted this way are gone and cannot be recovered.

Copying files

The command to copy a file is `cp` (**cp**y). It is followed by the file's present name and location (known as the source) and the location where you would like the copy to end up (known as the destination). In the following hypothetical example, the source is a file named `original.txt` and the destination is an identical file but with a different name, `duplicate.txt`:

```
host:~ lucy$ cp original.txt duplicate.txt
```

You can use `cp` to copy a file within the same directory, giving the new file a new name, as we just did. You can also copy a file to another directory, either giving it a new name or retaining the old name. These behaviors may seem different at first glance, but are actually quite similar and follow naturally from the ways that the source and destination are specified. If only the filenames are specified, everything takes place in the working directory. This is because a naked filename is just a relative path to a file in the working directory. If the source or destination is a path to a file in a different directory, then files can be moved between directories. The working directory can even be different than either the source or destination directory, provided that paths are specified for both source and destination. If the

When moving or copying a file, by default the shell will not warn you if a file or folder of the same name exists in the destination directory. If a file of the same name does exist, it will get "clobbered"—that is, deleted and replaced without warning or notification. You may not realize this until much later when you go looking for the original file. In Chapter 6 you will see how to modify this behavior.

source and destination directories are different, but the filenames are the same, then the file will be copied to the destination directory with the same name.

There is one other thing that may seem a little odd at first about `cp`, but saves quite a bit of typing in the long run: if you only specify a path to a directory for the destination, without an actual filename, then the file will be copied to that directory with its original name.

Remember that the command option consisting of two periods (`..`) refers to the directory that contains the current working directory. Similarly, a single period (`.`) represents the current working directory itself, which is useful if you want to copy a file from elsewhere to the working directory. In the following example, you will move to the `sandbox` folder and copy a file from the `examples` folder to it:

```
host:~ lucy$ cd ~/pcfb/sandbox
host:sandbox lucy$ ls  ← Probably nothing in this directory yet
host:sandbox lucy$ cp ../examples/reflist.txt ./
host:sandbox lucy$ ls
reflist.txt
```

The source, `../examples/reflist.txt`, in essence says "Go into the directory containing the working directory, then from there go into the directory `examples` and get the file `reflist.txt` within it." The dot followed by a slash, `./`, says "Place a copy of that file, with the same name, in the present working directory." Since only a directory is specified for the destination, with no accompanying filename, the copy of the file has the same name as the original, `reflist.txt`.

Now copy `reflist.txt` again, but to a new file with a different name in the same directory:

```
host:sandbox lucy$ ls
reflist.txt
host:sandbox lucy$ cp reflist.txt reflist2.txt
host:sandbox lucy$ ls
reflist.txt     reflist2.txt
```

Where there was once only one file, `reflist.txt`, there are now two files, `reflist.txt and reflist2.txt`. Their contents are exactly the same.

Moving files

The command for moving files is `mv` (**move**). You move files in the exact same way that you copy them, except that the original file disappears in the process. In the following example, you move the `reflist2.txt` file you just made to `Desktop`:

```
host:~ lucy$ cd ~/pcfb/sandbox
host:sandbox lucy$ ls
reflist.txt     reflist2.txt
host:sandbox lucy$ mv reflist2.txt ~/Desktop
host:sandbox lucy$ ls
reflist.txt
```

Remember that when you leave off the destination filename, it defaults to the same name as the original. If you now look on the Desktop you should see the `reflist2.txt` file there. Feel free to delete it if you like. You could do this with the command `rm ~/Desktop/reflist2.txt`.

The command `mv` is also used for renaming files, since renaming a file is equivalent to moving it to a new file with a different name in the same directory. The following example illustrates this:

```
host:~ lucy$ cd ~/pcfb/sandbox
host:sandbox lucy$ ls
reflist.txt
host:sandbox lucy$ mv reflist.txt reflist_renamed.txt
host:sandbox lucy$ ls
reflist_renamed.txt
```

In a later section of this chapter, you will see how to quickly move or copy multiple files at once.

Command line shortcuts

Up arrow

It is not too soon to introduce two shell shortcuts that will save you a lot of typing over the years. After all, the point of this book is to reduce the amount of extra work you have to do. Try to get into the habit of using these, because they will not only save you time, but will also reduce the number of typographical errors you have to correct.

The first shortcut is the ↑ key. In most shells, the ↑ moves back through your previous command history. For example, if you type in a long command and hit return, only to realize that you were in the wrong directory for the command to do what you wish, you can easily move to the correct directory, press ↑ one or more times until you see the correct command, and then hit return again to execute the command. Try pressing the ↑ key now to step back through and see all the `cd`'ing and `ls`'ing that you have done so far.[4]

[4] Pressing the ↓ will show future commands that you haven't typed yet.

Previous commands can also be edited before you re-execute them. If you enter a command but it has a typo somewhere in it, press ↑ to show it again, then use the ← and → keys to move to the place where the error occurred and fix it. Once you have corrected the typo and the command looks good, hit return. You don't need to use the → key to move the cursor back to the end of the line. When editing commands, depending on the terminal program and operating system, you can also change the position of the cursor within the terminal window using the mouse rather than the ← and → keys. Try it out now: press the ↑ key to bring up a command, and point with the mouse to where you would like the cursor to be inserted. Hold down the option key and click, and the cursor should be inserted at that point for quick editing.

Tab

Another big time-saver is the tab key. This is in effect the auto-completion button for the command line. For example, at the prompt, start typing:

```
host:~ lucy$ cd ~/Doc
```

Then, before pressing return, press tab. The command line should fill in the remainder of the word `Documents` and append a trailing slash. Now try this command:

```
host:~ lucy$ cd ~/D tab
```

You should get a beep or a blank stare from your terminal. This is because there is more than one folder starting with D in your home directory. The shell can't tell if you are referring to the Documents folder or the Desktop folder. To see what the choices are, press tab again. The terminal will return a list of the matches to what you have typed so far. If nothing shows up, check for incorrect slashes at the beginning, or else a missing tilde, to make sure you haven't entered the start of a path that doesn't exist.

Completion of directory and filenames is especially convenient when the name has a space in it. If you try to type a command with a space in it, such as `cd My Project`, the shell thinks that there are two different pieces of information (arguments) being sent to the `cd` command. Since you probably don't have a directory called My, it returns an error.

When you use tab auto-completion in this context, the shell types the command with extra backslashes inserted before the spaces:

```
host:~ lucy$ cd ~/My\ Project/
    If you type tab here...↵ ↳...the computer will complete the rest
```

Remember from Chapters 2 and 3 on searching and replacing with regular expressions that a backslash (\) modifies the interpretation of the character that follows. In this case, it causes the shell to interpret the space as part of the filename

rather than as the separation between two filenames. You don't need auto-completion to take advantage of \. You can also type it in yourself as part of the path.

Spaces can be accomodated in file or directory names by using either single or double quotes to enclose the full path, including spaces. However, you cannot do so if you are trying to use the tilde shortcut or wildcards as part of the path, since they won't be expanded.

It is generally best to just AvoidUsingSpacesAltogether when naming data files and scripts. This will make things easier later on. Other punctuation marks in names (`?`, `;`, `/`) have special meanings at the command line, and should therefore be avoided as well. Underscores (_) are acceptable to use.

Modifying command behavior with arguments

All of the commands we've discussed so far, such as `ls`, `cd`, and `pwd`, are actually little programs that are run by the shell. The programs each read in bits of information and do something as a result. Pieces of information that are passed to a program at the command line are called **arguments**. Each argument is generally separated from the program name and any other arguments by a space, but beyond that there aren't any global rules for how programs receive and interpret arguments. You've already used arguments, such as when you told `cd` which directory to change to, or when you told the `mkdir` program what directory to create. Some programs interpret arguments in a particular order, while others allow arguments to be in any sequence.

Besides a path name, the `ls` command has many optional arguments; in format, these consist of a hyphen followed by a letter. Two of the most commonly used arguments or options are `-a` and `-l`. Consider `-a` first. Some files and folders on the system, those whose names begin with a single period, are normally hidden from view. To tell the `ls` command to show all files, including those that are hidden, type `ls -a` (meaning list **a**ll). Try `ls -a` from within your home directory (or `ls -a ~/` from any folder), and you will notice some of the hidden files that show up. For example, the directory `.Trash` is normally a hidden folder into which your not-yet-deleted files are placed when you drag them to the Trash icon from within the graphical user interface. Other hidden files you see may be settings files used by the system. In a later chapter, you will edit an important hidden settings file called `.bash_profile`.[5]

You can also tell `ls` to return a more detailed list of directory contents with the `-l` argument. (This is a lowercase L, not the number 1.) This prints a list that includes, in columns from left to right, the file or directory permissions; the number of items in the folder, including hidden items; the owner; the group; the file

[5]These hidden files are not normally visible when working in the Finder or in other parts of the Mac GUI. To make them visible in the Finder, type the following in a terminal window: `defaults write com.apple.finder AppleShowAllFiles TRUE`. Then restart the Finder by logging out or typing: `killall Finder`. You can also go to hidden folders using the ⌘ shift G shortcut.

size; the modification date; and the file or directory name. We won't talk about permissions until later, but much of this information will eventually prove useful or necessary as you progress:

```
host:~ lucy$ ls -l
total 0
drwx------+  3 lucy  staff   102 Mar  9  2008 Desktop
drwx------+  4 lucy  staff   136 Mar  9  2008 Documents
drwx------+  4 lucy  staff   136 Mar  9  2008 Downloads
drwx------+ 30 lucy  staff  1020 Sep 28 13:07 Library
drwx------+  3 lucy  staff   102 Mar  9  2008 Movies
drwx------+  3 lucy  staff   102 Mar  9  2008 Music
drwx------+  4 lucy  staff   136 Mar  9  2008 Pictures
drwxr-xr-x+  5 lucy  staff   170 Mar  9  2008 Public
drwxr-xr-x+  5 lucy  staff   170 Mar  9  2008 Sites
drwxr-xr-x   3 lucy  staff   102 Apr 19 22:48 scripts
```

If you would like to both see a detailed view *and* all the hidden files, at the same time, use both arguments in the same command:

```
host:~ lucy$ ls -l -a ~/
```

This shows a detailed view that includes all the hidden files in your home folder. With many programs, you can combine arguments using one dash followed by the arguments together with no spaces; thus `ls -la ~/` is equivalent to the command above. The order of the arguments for `ls` isn't critical either, so `ls -al ~/` is also equivalent. See Appendix 3 for a more complete list of options.

Viewing file contents with `less`

There are several commands that display the contents of text files. This is useful when you have a huge file and want to take a quick peek inside, or just view a certain part of a file within the terminal without launching another program. The most commonly used file viewer is `less`.[6] Typing `less` followed by the path of a file presents the contents of that file on the screen one page at a time. Table 4.1 shows the keyboard shortcuts you can use to navigate in `less`. To move to the next page, type a space (that is, press the space key); to move back to the previous page, type `b`. Scroll up and down a line at a time with the arrow keys. To go to a

[6] If this seems hard to remember, know that Unix developers like to choose quirky names for their programs. The original file viewer was named `more`, and so when it was re-written to add functionality the name was changed to `less` because `less` is not `more`. If your system does not seem to have the program `less` installed, try `more` instead.

TABLE 4.1 Keyboard commands when viewing a file with `less`

q	Quit viewing	↑ or ↓	Move up or down a line
space	Next page	/abc	Search for text abc
b	Back a page	n	Find next occurrence of abc
## g	Go to line ##	?	Find previous occurrence of abc
G	Go to the end	h	Show help for `less`

specific line, type the line number followed by `g`. (Typing a capital `G` instead will take you to the bottom of the file.)

Try viewing some of your documents or data files with the `less` command, either by typing the full path of the file (remember to use tab auto-completion to make this easier), or by moving to the directory with `cd` and running `less` followed by the file name.

To find a word in the file you are viewing, type / followed by the desired word or character and hit return. If found, the line containing the word will be placed at the top of the screen. To find the next occurrence, type `n`.

When you have seen enough, type `q` to quit. These navigation shortcuts show up repeatedly in shell programs (though they are not universal), so it is worth remembering them as you work. They are summarized in Appendix 3 for quick reference. If you try viewing a file in the terminal and it appears on the screen as strange characters or gibberish, it is probably not a plain text file and has to be opened with a special program which understands how the binary data are laid out.

The program `less` does not load the entire file into memory before showing it to you; it loads only the part it is displaying. So if you have a 500 megabyte dataset and just want to see the very first part of it (perhaps to check that it contains what you think it does, or to figure out the column headers), you are much better off taking a look at it with `less` than you would be opening the entire file in a text editor (which could take some time and bog the entire system down). Later you will see ways to search inside a file without having to open it, using `grep`.

Viewing help files at the command line with `man`

Most command-line programs have user manuals already installed on your computer. These explain what the programs do, which arguments they understand, and how these arguments should be entered. These manuals are viewed with the program `man`, which takes an argument consisting of the name of the program you want to see the manual for. So, for instance, `man ls` will display lots of practical information about the `ls` program and the optional arguments it understands.

The `man` program uses the `less` program described above to display the help files, so you navigate within it using the same keys: space advances the file page by

page, a slash lets you find a certain word, and pressing `q` lets you quit the file and return to the command line when you are done reading.

Try using `man` to investigate some of the options for other programs you've already used, for example `man pwd` or `man mkdir`. Using `man`, you will be able to investigate any of the other programs which you encounter in coming chapters. Strangely, some of the most basic commands, including `cd`, `alias`, and `exit`, don't have their own `man` entries.

Many times you might not know the exact command that will help you do something, only the general topic. In this case you can search through the contents of all the manuals on your system using `man -k` followed by a keyword. For example, to find out commands dealing with the date, type `man -k date` and look through the brief descriptions of the commands that match. There are many utilities built in to Unix installations: calendar programs, calculators, unit converters, and even dictionaries.

The command line finally makes your life easier

Wildcards in path descriptions

So far you have not done much that you could not have accomplished just as easily (or maybe more easily) with a graphical user interface. It is the ability to use **wildcards** that suddenly makes the command line more powerful than a graphical user interface for processing many files in one operation.

While generating search queries back in the chapters on regular expressions, you encountered several types of wildcards—that is, shorthand ways to represent multiple characters. The shell has its own wildcards, but it can be confusing because it uses many of the same wildcard characters we have already encountered, but with significantly different meanings. The wildcard you will use most frequently in the shell is the asterisk (`*`). In the context of the shell it is the wildest of wildcards, representing any number of any kind of characters (except a slash). In regular expression language, this would be the equivalent of `.*` (the dot meaning any character, the asterisk meaning zero or more).

For example, to list all the files and folders that begin with `D` in your current directory, as well as the contents of those directories, you could type:

```
host:~ lucy$ ls D*
```

To list just files ending in `.txt`, you could type:

```
host:~ lucy$ ls *.txt
```

You can use several wildcards in the path. So to list all text files within all directories that start with `D`, you can type:

```
host:~ lucy$ ls D*/*.txt
```

Notice that the asterisk doesn't include text beyond the path divider (/). Therefore, `ls D*.txt` is not equivalent to the command above, because it would only show files in the current directory that start with `D` and end with `.txt`.

The construction of `*/*.txt` is a convenient way to indicate all the text files in all immediate subfolders of your current directory. It is even possible to search deeper folders, for instance with `*/*/*.txt`. These commands only show files at the exact specified number of levels, so they would not show a `.txt` file in the current directory.

When using wildcards, if `ls` finds a file that matches, it lists the filename; if it finds a directory that matches, it lists the directory along with its contents.

Wildcards carry the risk of causing problems through unanticipated matches. You especially want to be careful before using wildcards to remove, copy, or move important files; this will help keep you from accidentally modifying files you didn't realize were there. It is often wise to test wildcards with a harmless `ls`. This will give you a list of files recognized by the command so you can check to see if it includes any that you didn't anticipate.

Copying and moving multiple files

Using wildcards, you can quickly copy or move multiple files with names that match a certain pattern. In the following example, all of the files that end with `.txt` in the examples directory are copied to the sandbox folder:

```
host:~ lucy$ cd ~/pcfb/sandbox
host:sandbox lucy$ ls
host:sandbox lucy$ cp ../examples/*.txt ./  ← Very different from / by itself
host:sandbox lucy$ ls
reflist.txt  ← You will see others too...
```

This should begin to give you a sense of some of the tasks that can be easier working at the command line rather than in a graphical user interface. Both the copy and move commands, used in conjunction with wildcards, are significant timesavers when dealing with large numbers of files. You can imagine, for instance, how you could gather all your physiology data from a particular species by using the taxonomic name at the beginning and the data format at the end of a path (`Nanomia*.dat`) while ignoring any similarly named images or documents that might be in that folder.

Ending your terminal session

To end your session, type `exit`. You can just close the window, but this is a bit like unplugging your computer without turning it off first: if there are any programs still running in the terminal window, they will come unceremoniously to a halt. Closing the window is also not an option when you log in to a different computer from within your terminal. If `exit` does not work, some shells use `logout` or `quit`.[7]

[7] If none of those work, *then* pull the plug.

SUMMARY

You have learned:

- The difference between a GUI and a command line
- The difference between absolute and relative paths
- Ways of indicating relative paths, including:
  - `~` for home
  - `..` for the enclosing directory
  - `.` for the current directory
- How to move around in the terminal using `cd`
- Common commands, including:
  - `ls` to list the contents of a directory
  - `pwd` to print the name of the current working directory
  - `mkdir`, `rmdir`, `rm` to make and remove directories and files
  - `less` to view files
  - `cp`, `mv` to copy and move files
  - `man`, `man -k` to get help on a command or search for help
  - `exit`
- How to use the wildcard `*` in shell commands
- Command line shortcuts, including:
  - tab to complete program and path names
  - ↑ to recall prior commands in your history

Recommended reading

Kiddle, Oliver, Jerry D. Peck, and Peter Stephenson. *From Bash to Z Shell: Conquering the Command Line*. Berkeley, Calif: Apress, 2005.

Newham, Cameron. *Learning the Bash Shell*, 3rd Edition. Sebastopol, Calif: O'Reilly Media, 2005.

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
```

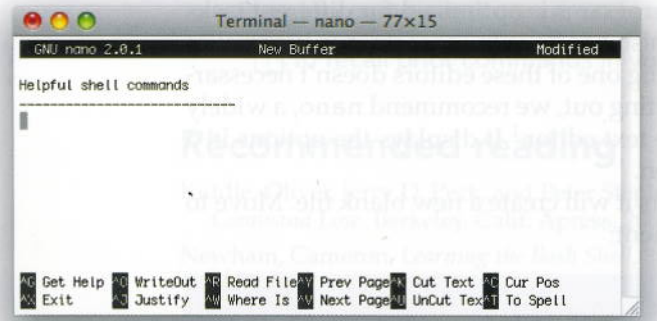

FIGURE 5.1 A view of the popular command-line text editor nano with the text you just entered Some of the available commands are displayed at the bottom of the window.

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.

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.

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.

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,

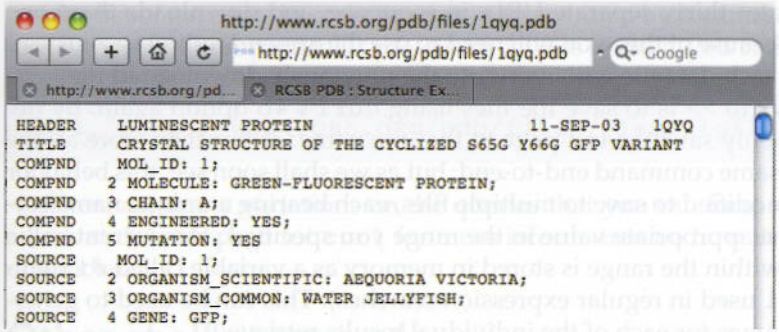

FIGURE 5.2 Example of a URL displayed in the address bar of a web browser

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`

Chapter 6

SCRIPTING WITH THE SHELL

You are now familiar with some of the powerful commands available within the shell, although to an extent you have been replacing one kind of work (clicking and dragging) with another (typing). You will now learn to gather multiple commands into a file that can be executed as a single command in the shell. This list of commands is a script and is itself a type of custom program.

Combining commands

A shell **script** is just a text file that contains a list of shell commands that you want to run in sequence. Certain properties of the file, which we will describe below, tell the shell how to interpret the commands in the file. When executed, usually by typing the name of the script at the prompt, commands in the file are executed as if you had typed each of them at the command line. Scripts are useful for situations where you want to use the same set of commands on many different occasions, or when you want to execute a set of many commands at once. Here are some examples of possible shell scripts:

- Download twenty files from the Internet and save them to a single large file
- Convert a file from one format to another, process it in a program, and reformat the output
- Rename a set of files or copy them to a different directory

All of these tasks can be accomplished using specialized programs as well as shell scripts, but sometimes the shell approach is the most direct.

Before you write any scripts, we will first walk through a few steps that are necessary for the shell to recognize a text file as a list of commands. In these and subsequent chapters, we will use the formatting convention that text to be entered

at the command line (in a terminal window) has `this style` and text in a script (a text editor document) has `this background`. As before, text that you type at the prompt is shown in bold.

The search path

How the command line finds its commands

Each of the commands we've shown you so far (`ls`, `cd`, `pwd`, etc.) is its own little program that the `bash` shell calls when you type in the program name. To find out the directory where a particular program is stored, use the `which` command. For example:

```
host:~ lucy$ which cd
/usr/bin/cd
host:~ lucy$ which ls
/bin/ls
host:~ lucy$ which which
/usr/bin/which
```

Even the `bash` shell that generates the command line is a program with its own location:

```
host:~ lucy$ which bash
/bin/bash
```

These folders (`/usr/bin` and `/bin`) are some of the standard directories where programs are stored on your system.[1] To list the pre-installed programs in these directories, type `ls /usr/bin` and `ls /bin`. If `which` gave you a different answer to the location of a command you tried, take a peek at the contents of that directory as well.

When you create your own script or program, how will the shell know where to find it when you type its name? One way is for you to provide the absolute path. For example, `/bin/date` runs the `date` command. However, it would be inconvenient to have to specify the full path to every program each time you wanted to use it. You also do not want to copy a program to your data directory each time you use it—this is a big mistake that people sometimes make. Copying a program to each directory with data files can lead to the same program being present in dozens of places, and it is a nightmare when a new version of the program comes out and you have to keep track of which version is where. *You want your computer to have a single working copy of each program, and no more.*

[1] Even though `/usr` looks and sounds like `/User`, it stands instead for Unix system resource. These commands are available throughout the system, not just to a particular user.

IMPORTANT The steps described here are critical for setting your computer up for scripting, as well as for programming and other tasks that will be described in later chapters. Follow along with them even if you haven't been trying out other examples.

For a variety of reasons, you don't want to put the programs you create in `/bin` alongside the system programs. First, your computer won't let you unless you grant yourself special administrative privileges. (We'll get to that later.) Second, you don't want to be mucking around deep within the core files of your computer like that. You could end up erasing an important command by mistake (where would you be without `ls`?) or altering things in such a way that they generate cryptic results. Better to make yourself a custom directory where you can put your own programs with less risk of damage, and then tell the shell to look for them there. This process is a bit complicated, but it only needs to be done once.

Creating your workspace, the scripts *folder*

The first step is to make a directory where your program files will go. To begin, `cd` to your home directory. Then use `mkdir` to create a folder called `scripts`. (Remember this command does the same thing as choosing New Folder... from the File menu in the Finder.) Type `ls` and look at the list of files to make sure that the new scripts directory is there.

```
host:~ lucy$ cd
host:~ lucy$ mkdir scripts
host:~ lucy$ ls
```

You can refer to the new scripts folder anywhere on your system using the full path (`/Users/lucy/scripts`) or starting with the shortcut to your home directory (`~/scripts`).

Although you now have a folder for your own custom scripts, your shell still doesn't know to look for programs there. The shell knows to look in `/bin` and `/usr/bin` because there is a settings file which lists the places programs may be found. When you log in, this list of places, along with other settings, is loaded into system-wide variables for you as a user. When you type something at the command line that might be a program name, the shell searches through this list to see if it finds anything in those locations that matches. The list itself is stored in a special shell variable called `$PATH`. To see the current contents of `$PATH`, type `echo $PATH`. (Remember, capitalization is important when working in the shell.) The `echo` command just echoes whatever input you provide it, which in this case is the contents of the variable `$PATH`.

```
host:~ lucy$ echo $PATH
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
host:~ lucy$
```

Your list may be longer or shorter than this; regardless, it shows all the places that the shell will try to find a command corresponding to what you have entered at the command line. You can see that each of the paths is separated by a colon, and that our old friends `/usr/bin` and `/bin` are there, in addition to several other directories.

You can take a look at other shell variables like `PATH` by typing `set`:

```
host:~ lucy$ set
BASH=/bin/bash
COLUMNS=80
HOME=/Users/lucy
HOSTNAME=hosts.local
LOGNAME=lucy
MACHTYPE=i386-apple-darwin9.0
OLDPWD=/Users/lucy/scripts
PATH=/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/
X11/bin:
PWD=/Users/lucy
SHELL=/bin/bash
USER=lucy
```

This listing only shows a subset of the variables that you will see. The contents of each of these variables is known to the shell and thus available at the command line, and can be retrieved by putting a `$` in front of the name. So if someone asks your name, instead of responding `lucy`, you could reply "Hi, I am `$USER`." Instead of typing `cd /Users/lucy` to move to your home directory, you could also type `cd $HOME`. In the upcoming section, we will use the contents of the variable `$HOME`, as well as `$PATH`.

Editing your `.bash_profile` settings file

We now need to show you how to permanently modify `$PATH` so it always includes your new scripts directory.

Your personal settings for the `bash` shell can be modified using a hidden file in your home directory called `.bash_profile`. This file is itself a script, and it contains a list of shell commands that are automatically run each time you log in or open a new Terminal window. The file probably will not exist on your system, but this depends on whether the default preferences have already been modified on your computer. To edit the file, make sure you are in your home directory and type:

For Linux, use `.bashrc` instead of `.bash_profile`.

```
host:~ lucy$ cd
host:~ lucy$ nano .bash_profile   ← Important step #1
```

Be sure to include the dot before the name and the underscore in the middle. The `nano` command will open an existing `.bash_profile` if one is present; oth-

erwise it will create a new empty document. If you prefer TextWrangler over nano, you can use the command `edit .bash_profile` to launch TextWrangler with the file displayed for editing.

Now put a new definition of `PATH` into this file that includes your scripts directory. Because you want to preserve the other paths too, you will create the new path by adding your scripts folder to the end of the existing paths. One curious property of variables like `$PATH` in the shell is that they are *read* with a `$` before the name, but they are set *without* the `$`. You will see what we mean in a moment.

Type this line into your file exactly as written:

```
export PATH="$PATH:$HOME/scripts" ← Important step #2
```

There must be no space on either side of the equals sign. Notice that the first instance of `PATH` does not have a dollar sign before it, because it is being *set*, not read. This command sets `PATH` to a combination of its old value (what you saw when you typed `echo $PATH`) plus a colon, plus the absolute location of your `scripts` directory. As a shortcut for creating the absolute path of the `scripts` directory, you read in the value of `$HOME`, which should be `/Users/lucy`, and add `/scripts` to the end of it. You could instead write out `/Users/lucy/scripts`, but then this `bash` profile might not work on another user's system. The whole expression after the equals sign is surrounded by straight quote marks. The `export` portion of the line just tells the shell to make this `PATH` variable available anywhere in the shell, not just inside the confines of this particular file.

For Linux or Cygwin, use `${HOME}` where it says `$HOME`.

If your path previously was `/usr/bin:/bin`, then after executing this command it will be `/usr/bin:/bin:/Users/lucy/scripts`. In other words, the shell will now include the scripts directory with the other places it looks for commands.

Once the `export PATH` command looks right, save the file. Because the filename begins with a dot, your `.bash_profile` will not be visible in a Finder window or with a normal `ls` command, even though it resides in your home directory. To list

---

CHANGING SYSTEM SETTINGS While you have your `.bash_profile` open, you have the option of turning on a safety setting. Remember how the `cp` and `mv` commands and the > redirect function will overwrite existing files without asking permission? You can turn on a warning to yourself by adding the following line to `.bash_profile`:

```
set -o noclobber ← Add this line when you edit your .bash_profile
```

This will make your system reluctant to "clobber" or wipe out existing files. You may sometimes work on systems that do not have this safety net, so you should not become dependent on it. However, especially when starting out, it is easy to have a typo cause you misery. At this point, then, we advise you to stay vigilant, but do turn on `noclobber` in your `bash` settings.

---

the file, type `ls -a ~/` in a terminal window, or type `cat ~/.bash_profile` to see its contents.[2]

Checking your new `$PATH`

The moment of truth: seeing if your new settings worked! The configuration file is only read when `bash` first launches, so you will need to open a new terminal window. In the new window, again type `echo $PATH`. This time you should see the absolute path of your scripts folder attached to the end of the default `$PATH`. If you see your `scripts` directory listed as part of the new path, congratulations! You are all set up now to start writing and using your own software. If you do not see a `scripts` directory in your path after creating a new terminal window, check for typos and then seek help in the forums at `practicalcomputing.org`. If you are working on an operating system other than OS X, also consult Appendix 1.

In the future, you can add other script and project directories to your path by appending them to the end of the `export` line, inside the quotes. It is a good idea, though, to only have a small number of such folders. It is not fun to spend time trying to find problems in a script, only to realize that the copy you are editing is not the same one that the shell finds and executes when you type the command. The order of directories that the shell searches is based on the order they are listed in your path, so you could have a script right in your current working folder, yet when you type its name at the command line, the version that is actually being executed is elsewhere on your computer.

At this point you have created a `~/scripts` folder, and edited your `~/.bash_profile` settings file so that the shell looks in this folder to find commands you type. This part of the setup only needs to be done once. Now you are ready to add as many scripts and other custom programs as you like to this special directory.

Turning a text file into software

You have a place to put your scripts, but how do you actually write a script to put in it? There are a few operations involved:

1. Type the commands into a file.
2. Tell the operating system which program it should use to interpret the commands.
3. Give the script the permissions it needs in order to be executed by the shell.

All that is required to make a working script is a text editor and a shell—both of which you are now familiar with.

Open up your text editor and enter the following text exactly as written (except for the note of warning, of course):

---

[2] If you use MATLAB, your `startup.m` file is similar to `.bash_profile`.

```
#! /bin/bash
ls -la ← Note: this is lowercase L, not the number one
echo "Above are the directory listings for this folder:"
pwd
echo "Right now it is :"
date
```

This file will print a complete directory, then echo a message, and then print the name of the working directory. It then prints the date. Save the file as `dir.sh` in your `~/scripts` directory, with the `.sh` indicating a shell file (Figure 6.1).

If you are editing in TextWrangler or in an advanced command-line text editor such as `emacs`, you will notice that the text colors change when you save the script. This is because when you save a file with a `.sh` extension, the editor realizes that it is a shell file and gives you color-coded hints about the content. You will see this behavior when working through the examples in other chapters as well, since TextWrangler is set up to recognize many types of program and data files.

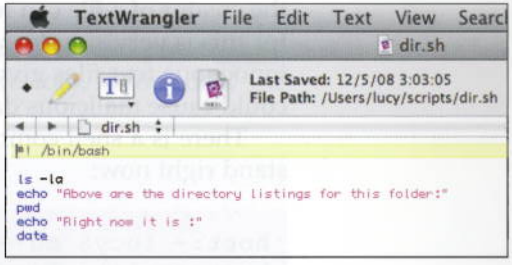

FIGURE 6.1 The `bash` script `dir.sh`

Control how the text is interpreted with #!

The first line of your script is very special. When the shell executes a text file and encounters `#!` at the beginning of the first line of a file, the shell will send the entire contents of the file to the program named immediately after `#!`. This means that you can write a file that automatically feeds a set of commands to any shell program. In this case, the program we are feeding the commands to is the shell itself, `/bin/bash`.

When you execute this text file a bit later, the shell will look at the first line, see `#! /bin/bash`, and send the rest of the contents to `bash`. You may have seen the hash mark (#) used to indicate comments in programs (that is, text to be ignored during execution), but it does something different when it occurs like this at the very top of a file.

The combination of `#!` is called a **shebang**, a name that comes from the "sh" in "hash mark" and the word "bang," which is a Unix-y way of describing the exclamation mark. It is sometimes hard to remember which order the characters go in (`!#` or `#!`), but if you remember shebang, you'll always get it right. The space after `#!` is not usually included, but we have used it here because it makes it easier to see the path of the file, and helps make sure you don't have an error (such as forgetting the leading slash and typing `#!bin/bash`).

You have now accomplished one of the key requirements to make a text file into a script: you've told the shell which program should interpret the contents of a particular file. This step—adding a `#!` to the beginning of the text file that contains a script—is necessary for each script you write, even in other languages.

Making the text file executable by adjusting the permissions

Try your new script by typing `dir.sh` and hitting return. You can do this from any working directory:

```
host:~ lucy$ dir.sh
-bash: /Users/lucy/scripts/dir.sh: Permission denied
```

It didn't work! This is because you still have to explicitly make the text file executable. The `#!` tells the shell what to do with the text when it is executed, but you still need to give the shell permission to execute the file as a command. By default, the creator of a file can read and write to the file, but cannot execute it. The reason for this is very simple: security. If any text file could be executed, then seemingly innocuous text files given to you by someone else or downloaded from the Web could cause malicious damage to your system.

There is a shell command to make a file executable. First, check where things stand right now:

```
host:~ lucy$ cd ~/scripts
host:scripts lucy$ ls -l ← Again, lowercase L
-rw-r--r--  1 lucy  staff      45 Dec 18 13:52 dir.sh
```

Remember that calling `ls` with `-l` causes it to list files and directories, one per line, and also to provide a bit more information about each listing. This information includes a compact representation of the permissions for the file (the dashes and letters at the beginning of each line), the owner of the file, the group for the file, the size of the file, the date it was last modified, and its name. Within the permissions section, the `r`'s indicate who can read to it, the `w`'s indicate who can write to it, and the `x`'s indicate who can execute it. These are grouped in order of who they apply to: user, group members, and all other users. A dash indicates that permission is not granted to that category of user. See Figure 6.2 for a closer look at how these are arranged.

In the permissions section of the listing for `dir.sh`, you may see that there are no `x`'s for now, which explains why the text file couldn't be executed as a program. You would need to modify the permissions of the file with the command `chmod` (**ch**ange **mod**e) to remedy this. You will do this with each script you write:

```
host:scripts lucy$ chmod u+x dir.sh
```

Check the results with another `ls -l` command:

```
host:scripts lucy$ ls -l
-rwxr--r--  1 lucy  staff  45 Dec 18 13:54  dir.sh
```

The permissions for `dir.sh` now include an `x`, indicating that the file is now executable by the user (in this case, you). The `u+x` argument for `chmod`, which comes between the command name and the filename, tells the shell, "For the main user/owner of this file, add executable permission." You can also subtract permissions using a minus in the first argument (e.g., `chmod o-x dir.sh`); this includes modifying the ability for others to read (`r`) and write (`w`) to a directory or file. Be careful not to take away your own permission to modify a file. For now, you will almost exclusively be using the `u+x` modifier.[3]

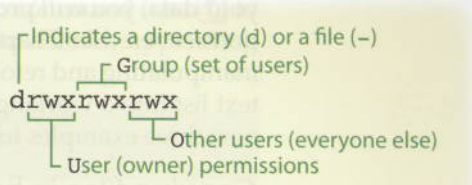

FIGURE 6.2 Permissions as shown by `ls`

Try your program again. Type `dir.sh` at the command line and press return. You should see a listing of all the files in your scripts directory, followed by the folder path and the current date:

```
host:scripts lucy$ dir.sh
total 8
drwxr-xr-x    3 lucy  staff  102 Dec 18 15:03 .
drwxr-xr-x+  19 lucy  staff  646 Dec 18 15:03 ..
-rwxr--r--    1 lucy  staff  118 Dec 18 15:03 dir.sh
Above are the directory listings for this folder:
/Users/lucy/scripts
Right now it is :
Sat Dec  18 15:03:34 PST 2010
```

You have created and run your first script! Now go somewhere else in your filesystem and try it again:

```
host:scripts lucy$ cd ~/Desktop
host:Desktop lucy$ dir.sh
```

By making the file executable, you have completed the final step of converting a text file to a script that can be run from anywhere on your system. This sets the properties of the file so that the shell knows it is safe to interpret the file contents as a set of instructions. You will need do this to make each script you write executable.

Generating scripts automatically

Now that you know how to make an ordinary text file into a script, you can begin writing programs that will help ease your workload. In the course of processing

[3] There are other ways to specify permissions using a binary-type format. See Appendix 6 for a more detailed explanation, or type `man chmod`.

your data, you will probably use regular expressions—the search and replace operations covered in Chapters 2 and 3—more than any other single tool. In addition to manipulating and reformatting datasets directly, regexp searches can convert a plain text list into a working script, with minimal drudgery. You will use this trick in the next three examples to create useful scripts.

Copying files in bulk

Imagine you have a folder with hundreds of files in it, and you want to copy a subset of those files to another directory based on their contents, rather than on their names. It is not possible simply to use `cp` with wildcards in this case; you will need to look within the files to see if they contain the text of interest, make a list of those files which qualify, and then use `cp` on that list.

In this example, you will write a script to copy only those files which contain the words `fluorescent` or `fluorescence` to another directory. The files in question will come from a series of 3-D structure files in PDB format, which are in your `examples` folder. Start by generating a list of the files you want—specifically, those containing the text fragment `fluor`. Use the `grep` command, with the option `-l` to return matching filenames rather than matching lines:

```
host:Desktop lucy$ cd ~/pcfb/examples
host:examples lucy$ grep -li fluor *.pdb
structure_1ema.pdb
structure_1g7k.pdb
structure_1gfl.pdb
structure_1xmz.pdb
```

Notice that by using the text fragment for our search word, we can match both `fluorescent` and `fluorescence` with the same query. Note too that in addition to `-l`, we added the case-insensitive flag, `-i`. The result is a list of all the files in `~/pcfb/examples/` with the extension `.pdb` and containing the text `fluor`.

The goal now is to reformat this list of filenames into a series of copy commands. If the list is short, you can copy it from the terminal window and paste it

---

PATHS AND GREP SEARCHES You can modify the scope of the `grep` search by using `*` in the path description. For example, to search inside `.pdb` files in all subfolders of a specific directory, you could write `grep fluor ~/pcfb/*/*.pdb`. The extra `*/` after `pcfb` means search *all* folders in the `pcfb` folder, not just the `examples` folder. This is a powerful technique to remember for the `ls` command as well. (For example, `ls ~/*/siphs*.fta`). When you add extra path elements in this way, though, the files must be located in the specified subdirectory. An asterisk here represents one or more subdirectories, not zero or more.

You may see an absolute or relative path in your results, depending on the way you call `grep`.

---

into a blank text document. If the list is long, rerun the `grep` command (use ↑ to step back through your command history) and redirect it into a file by appending `> ~/scripts/copier.sh`:

```
grep -li fluor *.pdb > ~/scripts/copier.sh
```

Instead of displaying the list to the screen, this will put the results into a file called `copier.sh`. Open this file in TextWrangler. (If you are not in the examples subdirectory, you can specify the full path as part of the `grep` command as `~/pcfb/examples/*.pdb`, but the results that are returned will also contain the full path, and not just the filenames by themselves.)

Here is where regular expressions come in. Instead of typing out the command for each filename, you will use regular expressions to semi-automatically transform each of these filenames into a copy (`cp`) command. These files are all located in the same directory, so the beginning of each copy command will be:

```
cp ~/pcfb/examples/
```

You want to insert this text without spaces before each filename. To do this, open the Find dialog, make sure Grep is checked, and search for the special boundary character ^ which, though not visible, indicates the beginning of each line. Remember that this marker represents the position just before the first character in a line. For the replacement text, type the `cp` command above. Figure 6.3 illustrates what your search box should look like.

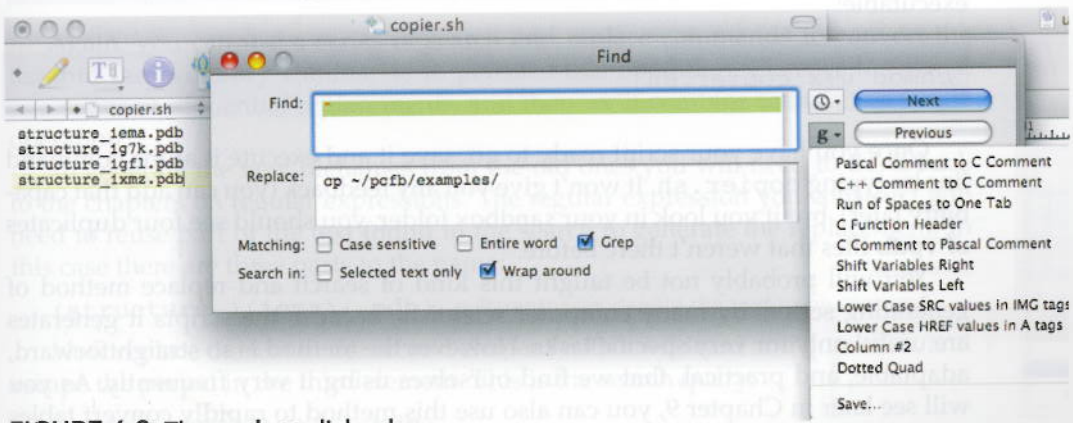

FIGURE 6.3 The replace dialog box displaying its Save Search menu g

After you perform this operation, you will have a file with the beginnings of several copy commands:

```
cp ~/pcfb/examples/structure_1ema.pdb
cp ~/pcfb/examples/structure_1g7k.pdb
cp ~/pcfb/examples/structure_1gfl.pdb
cp ~/pcfb/examples/structure_1xmz.pdb
```

Because the ^ searched for *any* line beginning, if you had a blank line at the beginning or end of your document, you will have an extra `cp` command in your file.

Now you need to append the rest of the copy command—the destination directory—to the end of each line. Search for the boundary character signifying the end of each line, `$`, and replace it with the name of the destination directory. In this case, you will copy the desired files to `~/pcfb/sandbox/`. Use this name, with a space before the initial tilde, as the replacement text. Each copy command is now complete:

```
cp ~/pcfb/examples/structure_1ema.pdb ~/pcfb/sandbox/
... lines omitted ...
cp ~/pcfb/examples/structure_1xmz.pdb ~/pcfb/sandbox/
```

This is a somewhat trivial example, because you could have pasted the commands together rather quickly. However, it is not hard to imagine cases where you are dealing with one hundred files instead of four. With two simple search and replace operations, you have turned a plain file list into a script.

At this point you should be protesting that it's not a working program quite yet. We have to indicate what kind of script this is by adding a shebang line (`#! /bin/bash`) at the beginning, and then change the permissions to make it executable:

```
chmod u+x copier.sh
```

Once you have your script ready to go, save it and execute it at the command line by typing `copier.sh`. It won't give you any feedback (you can add that capability later), but if you look in your `sandbox` folder, you should see four duplicates of `.pdb` files that weren't there before.

You will probably not be taught this kind of search and replace method of generating scripts by many computer scientists, because the scripts it generates are useful only for very specific tasks. However the method is so straightforward, adaptable, and practical, that we find ourselves using it very frequently. As you will see later in Chapter 9, you can also use this method to rapidly convert tables of information into program elements.

Flexible file renaming

Renaming files is generally more difficult than just moving or copying them to another directory. This is because `mv` and `cp` cannot automatically generate new filenames from portions of old filenames. For example, to rename a series of protein structure files ending in `.pdb` to files ending in `.txt`, you cannot just type `mv *.pdb *.txt`, since the `*` in the destination filename (`*.txt`) is not replaced with the characters matched by the `*` in the source filename (`*.pdb`). People have written a wide array of file-renaming utilities, but in this section you will learn to generate a shell script to rename an arbitrary list of files however you wish.

In TextWrangler, open the basic list of `pdb` files from the previous example. (You will need to regenerate this file if you already modified it.)

```
structure_1ema.pdb
structure_1g7k.pdb
structure_1gfl.pdb
structure_1xmz.pdb
```

This time, you will modify the filenames as you make new copies, removing `structure_` at the start and also changing the extension from `pdb` to `txt` at the end. When writing scripts which rename files, it is best to copy them with `cp`, rather than move and rename them with `mv`. Renaming is a risky process: even a single error in your script during the testing process could end up deleting all the files.

The basic command to copy a single file to one with the new name format would be:

```
cp structure_1ema.pdb 1ema.txt
```

Again, you can use a series of search and replace commands to convert the list into the necessary commands to perform this operation. First we'll show it as a series of sequential replacements, and then we'll combine them into a single operation.

To generate the new filenames from the old ones you will need to think back to the chapters on regular expressions. The regular expression you construct will need to reuse part of the text found in the search to generate the replacement. In this case there are three parts to the name:

```
(structure_)(1ema)(.pdb)  ← Parentheses denote the text to be captured
```

Now replace the text in the second element with a wildcard and quantifier, and escape the period in the third element. This is the search query:

```
(structure_)(\w+)(\.pdb)
```

Check the search term by doing Find without replacement. If it does not highlight the line, then check for trailing spaces or other inconsistencies in the format.[4]

All three parts are needed for the source filename, while only the second part, which falls between the underscore and the period, is reused for the destination filename. The replacement text that generates the destination filename will consist of the second element of captured text, (denoted by `\2`) plus the new file extension.[5]

```
\2.txt ← Backslash 2 is the contents of the second parentheses from the search
```

To create the entire replacement string in command format, add `cp` at the beginning, stitch back together the original filename with `\1\2\3`, and generate the replacement filename at the end with `\2.txt`:

Search term	Replacement term
`(structure_)(\w+)(\.pdb)`	`cp \1\2\3 \2.txt`

Run this replacement on the list of filenames, and you should have a series of command lines to rename files (move them) within a directory. You could either copy and paste these commands into a terminal window or save them into a script, as described below.

```
cp structure_1ema.pdb 1ema.txt
cp structure_1g7k.pdb 1g7k.txt
cp structure_1gfl.pdb 1gfl.txt
cp structure_1xmz.pdb 1xmz.txt
```

To turn this list of commands into a script, add the line `#! /bin/bash` to the top of the file, save it as something like `~/scripts/renamer.sh`, and then make it executable with `chmod u+x ~/scripts/renamer.sh`. As the script is currently written, nothing is specified about the paths except for the filenames; therefore, you must run it in the folder containing the files to be renamed. Note that the paths to the files are not relative to where the script is located, which is `~/scripts` in this and most other cases, but rather, relative to where you run the script (the working directory).

This script will only rename filenames that start with `structure_` and end with `.pdb`. If you wanted to write a single script that could handle filenames with other starts and endings, you could add further wildcards and quantifiers:

```
(\w+\_)(\w+)(\.\w+)
```

SAVING SEARCHES If you find a particular search to be especially useful or complex, you can save it in the TextWrangler Replace dialog by clicking on the pop-up menu labeled with a **g** next to the Find box (see Figure 6.3). The terms will then be accessible from any document you edit.

[4] If the concept of captured text is not familiar, review Chapter 2 on regular expressions.

[5] For advanced regular expression operators, it is possible to capture text in nested parentheses. The outer pair becomes `\1` and the inner pair becomes `\2`. So a simpler and equivalent search-replacement pair for this would be `(\w+\_(\w+)\.\w+)` replaced by `cp \1 \2.txt`.

In (somewhat) plain English, this matches `word_word.word` and captures each word in sequence as `\1`, `\2`, and `\3`. If you have names that contain punctuation (and especially periods), then your search would have to be adjusted to fit that situation.

Automating `curl` *to retrieve literature references*

Recall from Chapter 5 that the shell program `curl` retrieves documents over the Internet, much like a web browser does. This command already has the ability to retrieve a range of files at once, but it is still often useful to generate a script which contains a series of `curl` commands. This approach is more flexible than using `curl`'s wildcard ranges. In this case, we will start with a list of literature citations and from it generate a script to retrieve a more complete bibliographic record, including the DOI (digital object identifier, a universal address for electronic records). This script starts off with modest goals, but can potentially be made into a very useful reference retrieval device.

The CrossRef registration agency (`www.crossref.org`) provides a means of searching for published literature. The basic format for a CrossRef query is:

```
http://www.crossref.org/openurl/?title=Nature&date=2008&
volume=452&spage=745
```

The variable portions of the address have been highlighted here. This won't work quite yet, though. The system requires verification, so you can either register your e-mail address for free,[6] or temporarily use the a demonstration ID we have created. In addition, the default is for the URL to find the reference and redirect you to the journal's Web site, but we want to retrieve the full set of information associated with that reference. These can be gathered by adding more options to the URL, including a redirect field, a format field, the PCfB identifier for CrossRef:

```
http://www.crossref.org/openurl/?title=Nature&date=2008&volume=452&
spage=745&redirect=false&format=unixref&pid=demo@practicalcomputing.org
```

Open the file `~/pcfb/examples/reflist.txt`, and you will see that the file contains the Web address above, a series of reference entries, and the (very long) search and replacement strings you will use. Copy the first URL line (beginning with `http:`) into a browser's address bar (not the search box) and hit return. If it works, you can try plugging in some of your own search terms to see how customized queries can be formed. You will use this basic command to retrieve full reference information from several citations.

The references in the example file are in the format:

```
JournalName△   Year△   Volume△   StartPage
```
← Remember, △ indicates a tab character

[6] `http://www.crossref.org/requestaccount/` and see also `http://labs.crossref.org/site/quick_and_dirty_api_guide.html`

A nice feature of CrossRef searches is that author names, titles, and end pages are not required, and even the year is expendable. The actual entries are listed below:

```
American Naturalist 1880  14  617
Biol. Bull. 1928  55    69
PNAS  1965  53  187
Science     160  1242
J Mar Biol Assoc UK  2005  85  695
Biochem. Biophys. Res. Comm. 1985 126 1259
Gene  1992  111 229
Nature Biotechnology     17  969
Phil Trans Roy Soc B  1992  335  281
```

Cut and paste these reference lines into a new document, and save it with the name `getrefs.sh` in your `~/scripts` directory.

Now you will convert these entries into a `curl` command to retrieve the entries rather than view them one at a time. Because of the peculiarities of Web addresses (URLs), you will first need to replace any spaces that fall between parts of the journal names, using the replacement `%20`—that is, a percent marker followed by the ASCII code for the space symbol.[7] Search for spaces (not `\s`, and not tabs, but an actual space character) and replace all with `%20`.

Now generate a search query that will capture each of the four fields of the source file. Each field is separated by tabs; thus you can search for "any character" in the first field, and digits in the remaining fields. Some fields may be missing (for example, the year is missing in some references), but the corresponding tabs will still be there; because of this, you should use `\d*` instead of `\d+` for the first digit field, to allow for empty `\t\t` combinations:[8]

```
(.+)\t(\d*)\t(\d+)\t(\d+)
```

This search will store the journal name, year, volume, and starting page of the reference as `\1` through `\4` for use in the replacement.

To construct the replacement term, you can copy it from the example file:

```
curl "http://www.crossref.org/openurl/?title=\1\&date=\2\&volume=\3\&
spage=\4\&redirect=false\&format=unixref\&pid=demo@practicalcomputing.org"
```

This URL is so long that it is split onto two lines here, but in using it, make sure you keep it as a single unbroken line. Note that `\1` is located where the journal title should occur, `\2` is a placeholder for the year, and so on. The ampersands are escaped with backslashes to avoid being misinterpreted as captured text in the replacement string.

---

[7] See Appendix 6 for information on ASCII.

[8] Again, these regular expression search conventions should be familiar to you. If not, review Chapters 2 and 3 or Appendix 2.

When you perform this replacement, your list of bibliographic information will be transformed into a series of `curl` commands. Add a line containing `#! /bin/bash` to the start of the file, save it, and make the file executable with the command `chmod u+x getrefs.sh`. Try the script out by typing the name at the command line. It should print out a list of details about all the references. Once you have confirmed this, you can capture its output by redirecting to a file to retain your new bibliography:

```
host:sandbox lucy$ getrefs.sh > references.xml
```

The new `references.xml` file has much more information on each reference than the original file did. To find out how to automatically import this data file as a record in a bibliography program, look in the `reflist.txt` example file, or download the CrossRef importer plugin from `practicalcomputing.org`.

General approaches to `curl` scripting

For your own purposes in the future, you can take a general approach to making batch retrieval files:

1. Explore the Web site in a browser to find the exact search you are interested in. Note that in some cases it may be necessary to View Source for a page in your browser to find the actual link to data of interest.
2. From the address bar or page source, find the URL for a Web query that gives the result you want. Look for link names in the source, usually starting with `href=` . At times, these might have been visible from the preceding page, but not once you follow them to the data page itself.
3. Determine how the URL is constructed and what parts change. These changing elements will often come after a `?`, `=`, or `&` symbol.
4. Separate the URL into parts that can be used in your scripts. Sometimes the part that changes is just at the end, but sometimes there are several points where you will want to insert information.
5. Generate the regular expressions needed to convert the data you have into a `curl` command of the appropriate format.
6. Fold these commands into a `bash` script file and make it executable.
7. Save the output of your script with redirection >> if needed.

Aliases

You've seen how to join shell commands together into a script. There are also other ways to make multiple operations occur with a single command. One is to create a shortcut called an **alias**. With an alias, you can type a short simple command, and

have the system substitute a much longer command in its place. Aliases are defined using the following general format, with no spaces on either side of the equals sign:

```
alias shortcut="longer commands with options"
```

We mention aliases here because they can potentially save you time as you work through the upcoming chapters, but they are explained at length in Chapter 16. For example, you will probably spend a lot of time within the terminal window changing to your `scripts` or `pcfb/sandbox` directories. Even with the tab shortcut for completing directory names, this takes quite a few keystrokes. With an alias, you can make a shortcut which lets you easily change to that directory:

```
alias cdp='cd ~/pcfb/sandbox'
```

If you type this example at the command line, you will be able to type `cdp` at any time during your current terminal session and move to your `pcfb` folder from anywhere in the system.

Aliases defined at the command line only last until that terminal window is closed. To create a more permanent alias, edit `.bash_profile`, and add the alias line somewhere in that file. Each alias you create in this manner will be loaded for your use at the start of every session.

For other shells, see Appendix 1 for the format.

You can also use aliases to form the beginning of a command, as well as use them in combination with any terms that follow. For example, if you define this shortcut:

```
alias cx='chmod u+x'  ← Give a file executable permission
```

then you can just type:

```
cx myscript.sh
```

and the shell will respond as if you had typed out the entire `chmod` command. You can also use aliases with wildcards; in this case, for example, you can make all the `.sh` files in a folder executable with `cx *.sh`.

Other useful aliases include:

```
alias la="ls -la"  ← Directory listing with hidden files and permissions
alias eb="nano ~/.bash_profile"  ← Edit your .bash_profile
```

Later, as you find yourself working on a project or performing operations repeatedly, you can add your own personal shortcuts to your `.bash_profile`.

SUMMARY

You have learned how to:

- Set up your command line for scripting
- Manually write scripts specifying a list of commands
- Automatically generate scripts from a list of filenames or data
- Use `curl` in a script
- Use aliases to streamline commonly used command-line operations

Moving forward

Appendix 3 provides a reference table for the shell commands we have covered. Although we will be moving our focus away from `bash` commands for a while, more `bash` commands and ways to use them in your workflows are explained in Chapters 16 and 20. Other powerful shell commands you will encounter include `sort`, `uniq`, `cut`, and `wc`, which can be used in conjunction with each other and with other commands such as `grep`.