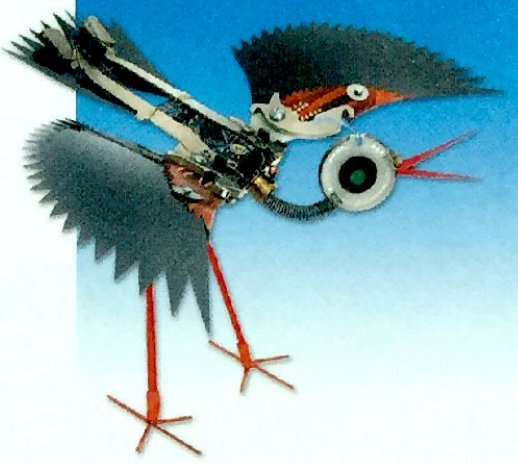
PART II



THE SHELL

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.

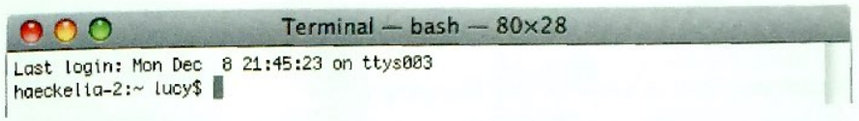


FIGURE 4.1 A portion of the Terminal window in OS X, showing the greeting, prompt, and cursor

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`.

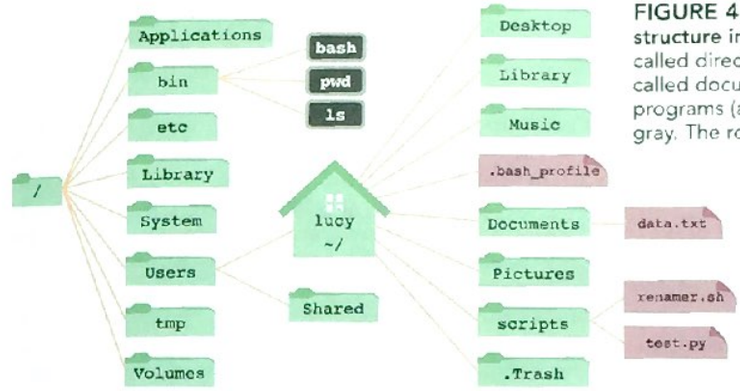


FIGURE 4.2 A portion of the filesystem structure in tree format. Folders (also called directories) are in green, files (also called documents) are in maroon, and Unix programs (also called commands) are in gray. The root directory is at the far left.

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`.

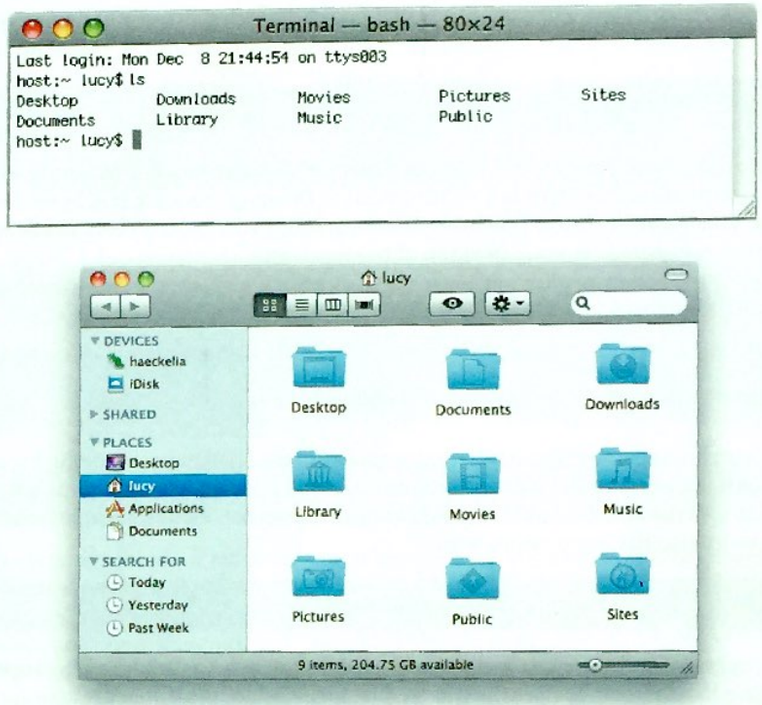


FIGURE 4.3 Viewing the interior of the home folder from two different interfaces, the Terminal window (top) and the familiar Finder window (bottom) Notice that the same folders are listed by both interfaces.

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
```

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.

On other systems you may not have a Desktop folder.

To see the absolute path of the working directory, use the command `pwd` (print working directory):

```
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` (change directory). 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 directory**).

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

[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`

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 time-savers 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.