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.

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.

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.

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; otherwise 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.

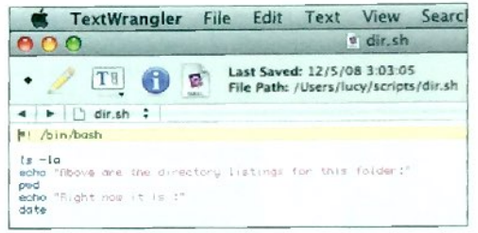

FIGURE 6.1 The `bash` script `dir.sh`

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]

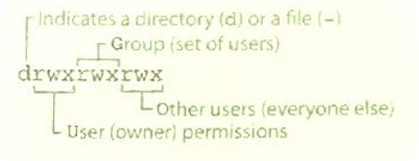

FIGURE 6.2 Permissions as shown by `ls`

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 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.

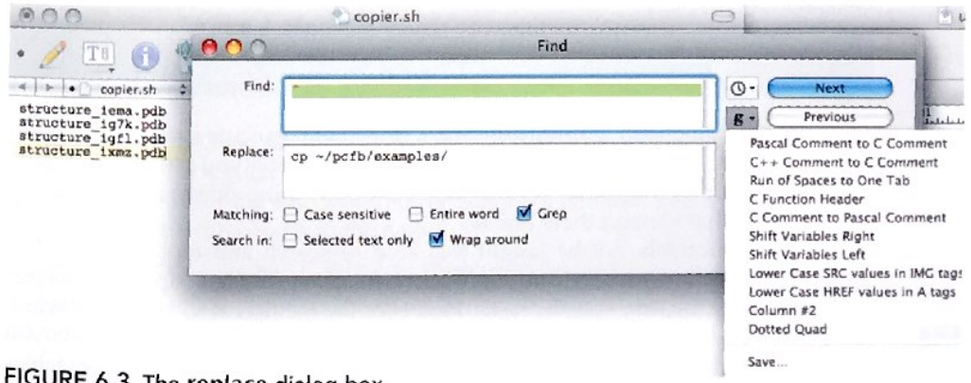

FIGURE 6.3 The replace dialog box displaying its Save Search menu g

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`.

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:

```
JournalNameΔ    YearΔ    VolumeΔ    StartPage  ← Remember, Δ 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`.