Chapter 16

ADVANCED SHELL AND PIPELINES

In Chapters 4 through 6 you became familiar with basic shell commands and with collecting them together in a file to create brief shell scripts. Here we return to the command-line environment and present new programs and operations, along with other ways to join commands together to form time-saving functions. We also show how to use shell operations to link together programs into data processing pipelines. These semi-autonomous scripts will help automate your analyses and keep a record of processing operations for future use.

Additional useful shell commands

Extract lines with `head` *and* `tail`

You have already seen how the `cat` and `less` shell commands will show the contents of a file. Two related commands are `head`, which shows the first lines, and `tail`, which shows the last lines of a file. These are used to get a quick glimpse into the contents of a long file, or, when added to the end of a command following the pipe symbol, to show just a portion of the printout of a shell command. Both `head` and `tail` can be modified to show X lines by adding `-n X` after the command name. For example, to see the first 5 lines of a data file:

```
head -n 5 ~/pcfb/examples/ctd.txt
```

Or, to see the 15 most recent items in your command history:

```
history | tail -n 15
```

Normally, the -n option used with `tail` will show you the last X lines. However, when a plus sign is added in front of X, it will show you the lines in the file starting with line X and continuing thereafter. For example, to skip a single header line:

```
tail -n +2 ~/pcfb/examples/ThalassocalyceData.txt
```

Both `head` and `tail` can accept more than one filename. The filenames can be specified either one after another, or via a wildcard—for example, `*.txt`. This makes for a convenient way to peek into many files at once.

TABLE 16.1 Options for the `cut` command

`-f 1,3`	Return columns 1 and 3, delimited by tabs
`-d ","`	Use commas as the delimiters, instead of tabs; this flag is used in conjunction with the `-f` option
`-c 3-8`	Return characters 3 through 8 from the file or stream of data

Extract columns with `cut`

The `grep`, `head`, and `tail` commands slice out lines of interest, extracting horizontal selections from a file. To pull vertical columns from a file, on the other hand, you can use the `cut` command (Table 16.1). This operates based either on character position within the column when using the `-c` flag, or on delimited fields when using the `-f` flag. With the `-c` flag, numbers are given to indicate which characters to extract; with the `-f` flag, the numbers indicate which columns to extract.

To pull out the first few characters of lines containing the > character without displaying the > itself, you can use:

```
grep ">" ~/pcfb/examples/FPexamples.fta | cut -c 2-11
CAA58790.1
AAZ67342.1
ACX47247.1
ABC68474.1
AAQ01183.1
```

The `cut` command is most often used to extract columns of data delimited by spaces, tabs, or some other character, using the `-f` option. The number following `-f` indicates which field, list of fields, or range of fields to retrieve:

```
cut -f 2-4 ~/pcfb/examples/ThalassocalyceData.txt
Depth   Latitude  Longitude
348.7   36.71804   -122.0574
520.3   36.749134  -122.03682
118.36  36.83848   -121.96761
200.2   36.723267  -122.05352
100.85  36.726974  -122.04878
1509.6  36.584644  -122.52111
```


By default, `cut` expects tabs as the delimiter; it will not split on spaces without the use of the `-d` flag followed by a space between quote marks. Bear in mind that a consecutive sequence of spaces will not be treated as a single separator, but as multiple separators. Multiple spaces are often used, for instance, to justify columns of text that have entries with variable numbers of characters. You may need to reformat data to accommodate this behavior. You could, for example, open the file in your text editor and use a regular expression to replace one or more spaces, as designated with `" +"`, with a single space, `" "`.

To indicate a comma as the delimiter, use `-d` followed by a comma, again with quotes around it, as is safest for shell operations whenever you want a punctuation mark interpreted as a string:

```
head -n 20 ~/pcfb/examples/ctd.txt | cut -f 5,7,9 -d ","
depth,temper,salin
2.78,15.299,33.132
3.47,15.3,33.133
8.64,15.298,33.134
15.29,15.295,33.134
21.84,15.003,33.155
...
88.93,10.614,33.439
95.28,10.403,33.464
101.53,10.188,33.486
107.68,10.03,33.539
```

Sorting lines with `sort`

The output stream produced by any of these commands, as well as the lines of a file or of a series of files, can be sorted into alphabetical order with the built-in `sort` command (Table 16.2). By default, sorting starts with the first character of the line and the first column of data:

```
grep ">" ~/pcfb/examples/FPexcerpt.fta | sort
>Anthomed
>Avictoria
>BfloGFP
>Pontella
>ccalRFP1
>ccalYFP1
>ceriOFP
>discRFP
>ptilGFP
>rfloGFP
>rfloRFP
>rrenGFP
```

TABLE 16.2 Options for the `sort` command

`-n`	Sort by numeric value rather than alphabetically
`-r`	Sort in reverse order, z to a or high numbers to low numbers
`-k 3`	Sort lines based on column 3, with columns delimited by spaces or tabs
`-t ","`	Use commas for delimiters, instead of the default of tabs or white space
`-u`	Return only a single unique representative of repeated items

To sort based on other columns (whether separated by tabs or spaces), use the `-k` flag (possibly standing for "kolumn"?) followed by the number of the column you wish to use for sorting. Note that because values are sorted in ASCII order (see Appendix 6), blanks come alphabetically before the letter `A`, so if there are empty values in a column of a tab-delimited file (in other words, two tabs in a row), those fields will be sorted to the top of the list. Another behavior is that all capital letters come before lowercase letters, so capital `Z` is alphabetically before lowercase `a`.

Sorting can proceed numerically instead of alphabetically when the `-n` flag (numeric) is used. Compare the output of commands sorting by the second column of data, first without and then with `-n`:

```
tail -n +2 ~/pcfb/examples/ThalassocalyceData.txt | sort -k 2
Thalassocalyce  100.85  36.726974   -122.04878  2.30    1999-08-09
Thalassocalyce  118.36  36.83848    -121.96761  1.52    1999-05-14
Thalassocalyce  1509.6  36.584644   -122.52111  0.95    2000-04-17
Thalassocalyce  200.2   36.723267   -122.05352  1.63    1999-08-09
Thalassocalyce  348.7   36.71804    -122.0574   1.48    1992-03-02
Thalassocalyce  520.3   36.749134   -122.03682  0.52    1992-05-05

tail -n +2 ~/pcfb/examples/ThalassocalyceData.txt | sort -k 2 -n
Thalassocalyce  100.85  36.726974   -122.04878  2.30    1999-08-09
Thalassocalyce  118.36  36.83848    -121.96761  1.52    1999-05-14
Thalassocalyce  200.2   36.723267   -122.05352  1.63    1999-08-09
Thalassocalyce  348.7   36.71804    -122.0574   1.48    1992-03-02
Thalassocalyce  520.3   36.749134   -122.03682  0.52    1992-05-05
Thalassocalyce  1509.6  36.584644   -122.52111  0.95    2000-04-17
```

When sorting is done alphabetically, the value `1509.6` falls between `118` and `200`; however, when it is done numerically, this same value is placed at the end of the list where it belongs.[1]

Isolating unique lines with `uniq`

Another powerful and frequently used command for extracting a subset of values from a file, or summarizing a stream of text, is `uniq` (Table 16.3). This command removes consecutive identical lines from a file, leaving one unique representative.

[1] Graphical interfaces for different filesystems don't all sort numbers in filenames the same way: some sort alphabetically, while others sort numerically. Now and then it can pay to know which of these two methods your particular operating system favors.

In order to be removed, the matching lines have to occur in immediate succession, without any intervening different lines. To get a single representative of each unique line from the entire file, in most cases you would need to first sort the lines with the `sort` command to group matching lines together. (The `sort` command actually includes a flag to return only unique records (`-u`), but `uniq` has other capabilities that make it useful in its own right.)

TABLE 16.3 Options for the `uniq` command

`-c`	Count the number of occurrences of each unique line
`-f 4`	Ignore the first 4 fields (columns delimited by any number of spaces) in determining uniqueness
`-i`	Ignore case when determining uniqueness

The `uniq` command can be used with the `-c` flag to count the number of occurrences of a line or value as it consolidates them. This gives a quick way, for example, to assess the number of occurrences of each taxon in a data file, or the most common annotations in a table.

Combining advanced shell functions

The output from a shell command can be sent to the screen or captured to a file. It can also be piped into another shell command, forming a chain of operations that can distill heavily processed values from a complex file. In this example, the starting point is a PDB file that describes the three-dimensional position of each atom and amino acid of a protein. We will build a combined shell command using `cut`, `sort`, and `uniq` to extract a table from this complex file, showing the frequency of occurrence of each amino acid.

To begin, move into the examples directory and take a look at the first few lines of each of the `*.pdb` files, using the `head` command:

```
host: lucy$ cd ~/pcfb/examples
host:examples lucy$ head -n 2 *.pdb
==> structure_1ema.pdb <==
HEADER  FLUORESCENT PROTEIN                             01-AUG-96   1EMA
TITLE   GREEN FLUORESCENT PROTEIN FROM AEQUOREA VICTORIA

==> structure_1g7k.pdb <==
HEADER  LUMINESCENT PROTEIN                             10-NOV-00   1G7K
TITLE   CRYSTAL STRUCTURE OF DSRED, A RED FLUORESCENT PROTEIN FROM

==> structure_1gfl.pdb <==
HEADER  FLUORESCENT PROTEIN                             23-AUG-96   1GFL
TITLE   STRUCTURE OF GREEN FLUORESCENT PROTEIN

==> structure_1s36.pdb <==
HEADER  LUMINESCENT PROTEIN                             12-JAN-04   1S36
TITLE   CRYSTAL STRUCTURE OF A CA2+-DISCHARGED PHOTOPROTEIN:

==> structure_1sl8.pdb <==
HEADER  LUMINESCENT PROTEIN                             05-MAR-04   1SL8
TITLE   CALCIUM-LOADED APO-AEQUORIN FROM AEQUOREA VICTORIA
```

```
==> structure_1sl9.pdb <==
HEADER  LUMINESCENT PROTEIN                          05-MAR-04   1SL9
TITLE   OBELIN FROM OBELIA LONGISSIMA

==> structure_1xmz.pdb <==
HEADER    LUMINESCENT PROTEIN                        04-OCT-04   1XMZ
TITLE     CRYSTAL STRUCTURE OF THE DARK STATE OF KINDLING FLUORESCENT
```

Note that the filename, placed between ==> and <==, is specified before each pair of lines since a wildcard is used to specify more than one file.

The following example will use the file `structure_1gfl.pdb`, but the resulting command can be applied to any of the other PDB files as well. Open the `structure_1gfl.pdb` file from the examples folder in a text editor to see the contents. You can interact with a 3-D rendering of the structure online at `tinyurl.com/pcfb-gfp`.

The protein in the example file is a dimer, having A and B subunits of the same molecule. We will create the shell command so that it only gives the results from one of the subunits, but you could just as easily have it return the count of all amino acids from both subunits of the molecule.

Notice that there are many introductory lines and remarks at the top of the file, as well as a few lines at the end, which do not contain the amino acid information we need. In the first pass at isolating the information we do need, we will extract only lines containing the word ATOM using a `grep` command (Figure 16.1). This eliminates many of the irrelevant lines, but leaves in remarks which contain the word ATOM. To remove these particular remarks, pipe the output of the first `grep` to an inverted `grep -v` command, which returns only lines that do not contain the word REMARK:

```
grep ATOM structure_1gfl.pdb | grep -v REMARK
```

Note that you only have to indicate the name of the file in the first command, since the subsequent commands operate on the output of the previous command. In this way you create an ever-more-reduced set of results—in this case, all the lines of the file that contain the word ATOM and not the word REMARK. The maroon area at the top of Figure 16.1 shows an excerpt of this stage.

At each point in this process, to see more clearly the results of the intermediate steps, you can append `| less` or `| head` to the end of the command. This way you won't see all the lines of output, just the first or last few.

After the two `grep` commands, each line of output now contains an atom associated with an amino acid (ALA at the beginning, THR at the end) and the sequential number of that amino acid. You can see that each amino acid is listed across several lines because each of its atoms is listed, but we will want to remove these repeated entries and just leave one line representing each amino acid.

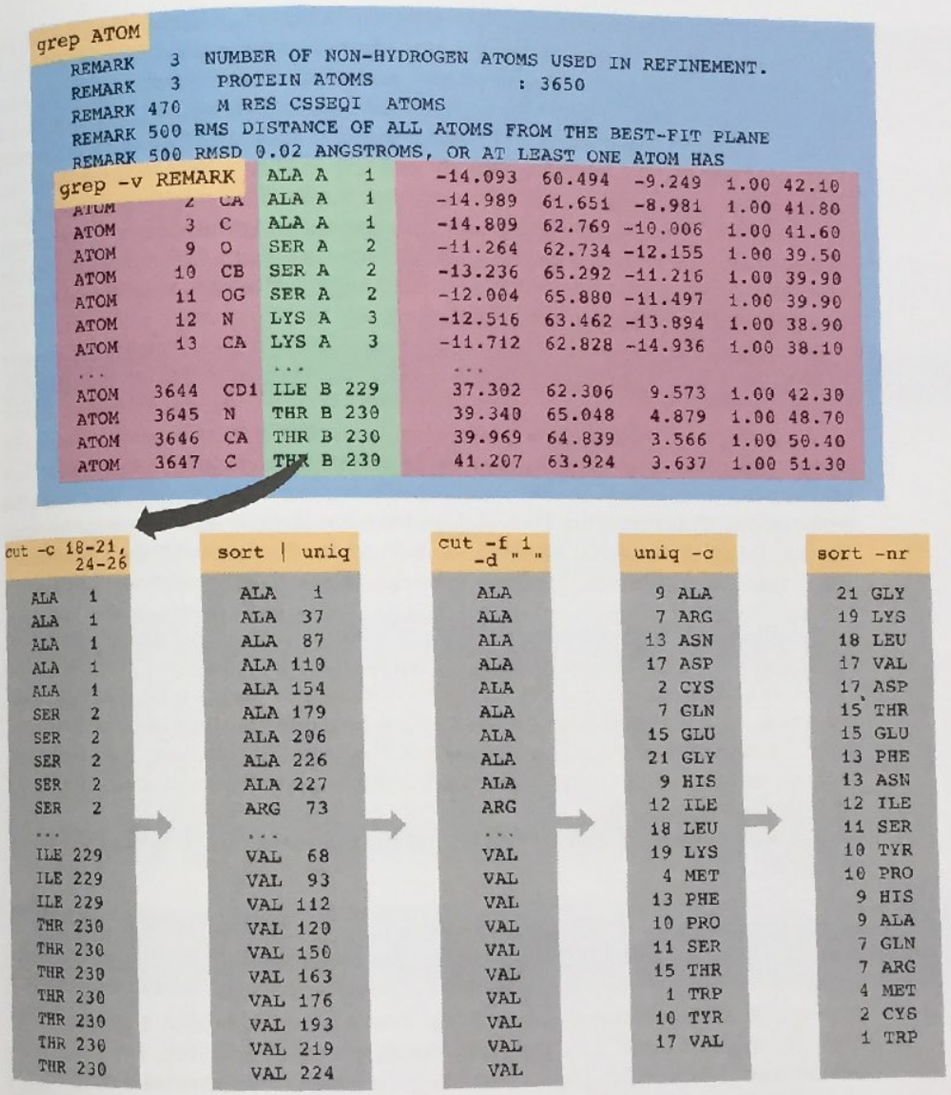

FIGURE 16.1 The successive extractions and modifications made by each command in the example pipeline Orange boxes show bash commands and other boxes show the output once those commands have been added to the pipeline.

The first step in doing this will be to use the `cut` command to extract just the amino acid three-letter code and the numerical position, characters 18 to 21 and 24 to 26. (The intervening A or B indicates which repeated subunit includes that

amino acid.) When you create the `cut` command, as we will do next, be sure not to leave any spaces between the numbers, dashes, or commas.

Pipe the output of the two `grep` commands into the `cut` command to get the results shown in the first gray column of Figure 16.1. At this point, each amino acid is represented by a few identical lines, for example `ALA 1` or `THR 230`. Because of the `A` and `B` subunits, there are actually two sets of `ALA 1` lines in the output, one at the beginning and one near the middle of the list. To bring these next to each other, we'll first `sort` the output and then use the `uniq` command to remove all repeated instances of each amino acid. (The number next to the name prevents all occurrences of a particular amino acid from being reduced to a single mention at this point.) The command so far consists of:

```
grep ATOM structure_1gfl.pdb | grep -v REMARK | cut -c 18-21,24-26 |sort|uniq
```

Now there is one line for each unique occurrence of an amino acid in the file. For instance, the nine lines of `ALA` at the top of the list indicate that there are nine total alanines in the sequence. To add up all of these occurrences, you will first cut out the first column, thus removing the numbers from the lines. In this case, instead of using `cut` based on character position, we will extract the first field (the first column) using space as a delimiter, by piping the output through this additional function:

```
| cut -f 1 -d " "
```

At this point we have a list of amino acid names that can be counted using the `uniq -c` command. This function will return two columns of data: first, how often an element is repeated, and second, the name of the element. As a final step, you can sort this list in reverse numerical order to place the most abundant amino acids at the top of the list.

The final command is given below, and the output is shown in the last column of Figure 16.1:

```
grep ATOM structure_1gfl.pdb | grep -v REMARK | cut -c 18-21,24-26 |
 sort | uniq | cut -f 1 -d " " | uniq -c | sort -nr
```

At the end of this chapter you will see how to turn this into a general function, so that you can just type the function name followed by a filename (`countpdb structure_1gfl.pdb`) to get a table of this processed output.

Approximate searches with `agrep`

By now, you are familiar with the `grep` command, which lets you search for lines within a file which either match or don't match a particular string. However, sometimes it is helpful to be able to search for a *nearly* exact match of a string. This can be achieved using a specially modified command called `agrep` (approximate `grep`), which lets you search for matches that include a number of substitutions, matches to entities that span more than one line, and matches to two alternate strings (Table 16.4).

Unlike `grep`, `agrep` is not installed by default. It can be downloaded for all major platforms using the appropriate link at the bottom of `en.wikipedia.org/wiki/Agrep`. (For OS X, choose the Unix command-line version.[2])

TABLE 16.4 Some options for `agrep`*

`-d "X"`	Use `X` as the delimiter between records rather than the end-of-line character
`-B -y`	Return the best match, without specifying an exact number; `-y` tells it to print this best match without asking
`-2`	Return results with up to this many mismatches between the query and the record; the maximum allowed is 8
`-l`	Only list filenames that contain a match
`-i`	Case-insensitive search

*See `man agrep` for a complete list

One use for `agrep` is to search through many protein or DNA sequences stored in a FASTA file to retrieve a short sequence which is a close match to a query sequence. Recall that FASTA files have sequence names on lines beginning with > followed by lines that contain sequence information.

The command `agrep -d "\>"` indicates that you wish to search in text blocks divided by > instead of by line endings. Instead of lines that match the specified term, the output will now be multiline blocks that contain that term. To avoid interpretation as a redirection symbol, the > character is in quotes as well as escaped with \. When a match is found using this delimiter, `agrep` outputs the sequence name and full sequence, rather than just the single line where the match occurred.

Although you can specify an allowed number of mismatches using a number as a flag, you can also have `agrep` return the best match or matches by adding two additional flags, `-B -y`.

To find the sequence with the best match for the amino acid fragment `CYG` in the file `FPexcerpt.fta`, for example, use the command:

```
agrep -B -y -d "\>" CYG ~/pcfb/examples/FPexcerpt.fta
```

Additional `grep` tips

Because the tab key has several functions at the command line, it is difficult to use it as a search term. In such cases, you can use the ctrl V operator. When you press and then release ctrl V at the command line, it causes the shell to interpret the keypress that follows literally, rather than carrying out its operation. Even the delete and return keys will be converted into their text representations when they come after the ctrl V sequence. To see that the return key is equivalent to ctrl M (^M),[3] try typing:

Your keyboard may say backspace

```
echo "ctrl v return"   ← In this case, you do type the quote marks before and after the other keystrokes
```

[2] See Chapter 21 for instructions on installing `agrep`.

[3] Recall from Chapter 5 that a caret (^) before a given character indicates that you hold down the ctrl key while pressing that character.

Using this method, in order to use tabs in a `grep` search, you can type "ctrl-v tab" and it will insert an invisible tab character into your search string.

Searching for negative numbers with `grep` is trickier than it might seem, because the character that follows a dash is usually considered some kind of modifier argument. For example, if you try to find all the lines containing `-8` with `grep -8` or `grep "-8"` or even `grep \-8`, it won't work. For this search to work, you have to use all your tricks and search with a quoted and escaped version:

```
grep "\-8"
```

Counting words and lines The `bash` command `wc` followed by a file name can be used to print out a count of the lines, words, and characters in a file. When placed by itself after a pipe symbol, `wc` will quantify the components of whatever output is piped to it.

Remember aliases?

At the end of Chapter 6 we gave a brief introduction to shell aliases. These are little shortcuts for commonly used commands. In this section, we present a few especially useful aliases. These examples will help demonstrate the syntax for defining aliases of your own, either at the command line or in your `~/.bash_profile` settings file. Defining aliases at the command line is a good way to create and test them, but unless defined in your settings file, they won't survive once you close your terminal window or log out of a session. If you have an account on a remote machine, you will want to define your favorite aliases in your settings file for that machine as well.

Our first example creates a shortcut for a command which prints a list of the ongoing processes, and then filters this list so that it shows only those processes which include your user name—in this case, `lucy`.[4] (One of those processes will be `grep lucy` itself!) This alias is convenient for finding runaway processes and halting them:

```
alias myjobs="ps -ax | grep lucy"   ← Show the processes matching user lucy
```

This next alias is useful if you regularly log into a remote machine using the `ssh` command (see Chapter 20 for details). You can create a shortcut of just a few characters, so as to save you from typing the full command each time:

```
alias sp='ssh -l lucy practicalcomputing.org'
```

Even the example for `agrep` described in the previous section can be turned into an alias:

[4] The `ps` command is discussed at some length in Chapter 20.


```
alias ag='agrep -B -y -d "\>" '
```

Notice that single quotes are used to set the boundaries of the alias definition, with double quotes nested inside to define the delimiter. This alias can then be called using the command:

```
ag CYG ~/pcfb/examples/FPexcerpt.fta
```

where `CYG` is the query sequence you are trying to find, and `FPexcerpt.fta` is the FASTA-formatted data file to search.

Many times you will want to look back through your recent commands to re-execute or copy one of them. With the alias shown here, you can type `hg pcfb` to find all the occurrences of the string `'pcfb'` in your recent command history:

```
alias hg='history | grep '
```

Finally, to see a list of your currently defined aliases, you can type `alias` by itself. These alias shortcuts, while useful for frequently used operations, are limited in how complicated they can be. Not to worry: you'll learn next about shell functions, which allow you to write miniature programs within the shell to accomplish repeated tasks with ease.

Functions

To create smarter multiline commands than those possible with aliases, you need to use a different style of shortcut, called a **function**. The syntax of shell functions is relatively cryptic, so we will limit this discussion to showing you the most useful basics and providing a few examples.

Functions are defined using the following format, with each such definition being added to your `.bash_profile`:

```
myfunction() {
  first command
  second command
  last command
}
```

The first part of the definition is the function name—that is, what you will type to run it. This is followed by `()` and an open curly bracket. You can also begin a function definition with this syntax:

```
function myfunction {
```

To replicate the functions of the `dir.sh` script you wrote in Chapter 6, for example, you could add the same commands to a function definition, here called `listall`:

```
listall(){
  ls -la
  echo "Above are directory listings for this folder: "
  pwd
  date
}
```

You could of course define this function by typing or pasting these lines at a shell prompt—but as with aliases created in this manner, the effect will only last until that shell session is closed. To have functions available each time you open a new shell window, you must add their definitions to your `~/.bash_profile` (or `.profile` or `.bashrc` for Linux). In your `.bash_profile`, the definitions of these functions are written in the same manner as if they would be typed at the command line, so there is no need to include a `#!` line as you do with shell scripts.

All of the functions described in this section are available in the file `~/pcfb/examples/scripts/shellfunctions.sh`. Within shell functions, the `echo` statement is the equivalent of `print` in Python. This is used here in the `listall` function to print out a plain string of quoted text. The shell can also have variables, traditionally named with all capital letters. Remember in Chapter 6 how you set the value of the `PATH` variable:

```
export PATH="$PATH:$HOME/scripts"
```

In that case, you were setting a new value of `PATH` while reading the existing values of the variables `$PATH` and `$HOME`. In the shell, a variable name with a `$` is *read from*, while the same name without a `$` is *assigned* a value.

At the command line, try typing:

```
echo $HOME
```

This is the command-line way of saying "Print the value of the variable `HOME`." System-wide variables such as `HOME`, `PATH`, and `USER` are available within your shell functions. You can also use the special variables `$1`, `$2`, and `$@` to represent user arguments that follow your command. `$1` is a variable containing the first user-specified argument, `$2` is the second one (separated from the first argument by a space), and `$@` contains all the arguments together.

Create a simple function by typing these next lines in a shell window. Notice that you see a `>` at the prompt after having typed the first line; this indicates that the shell is waiting for you to finish your function definition with a close bracket:


```
repeater(){
 echo "$1 is what you said first"
 echo "$@ is everything you said"
}
```

With these lines, you have created a miniature shell script that is able to operate based on user input. Test out your function at the command line with some varied input:

```
host:~ lucy$ repeater 1 2 3
1 is what you said first
1 2 3 is everything you said
host:~ lucy$ repeater "1 2" 3
```

Notice that this works exactly like the `sys.argv[]` variable in Python, as described in Chapter 11.[5]

An immediate use for this would be if you had a frequently used shell command where the filename was followed by a long string of additional values. Typically, only the filename changes each time you use this command, but you can't easily use the ↑ key to edit the command line, because the change is not at the end.

For example, `phyml` is a program that infers maximum-likelihood trees from genetic sequence files. It takes the data filename first, followed by a long string of options:

```
phyml sequencesA.fta 1 i 1 100 WAG 0 8 e BIONJ y y
```

When using this command again on different datasets, you would specify the same options, but the sequence filename would change with each use.

For greater convenience and less typing, wrap the command within a function definition and replace the filename with `$1`:

```
myphyml(){
 echo "myphyml $1 performs 100 bootstraps on AA data"
 phyml $1 i i 1 100 WAG 0 8 e BIONJ y y
}
```

[5] Similar to Python's `sys.argv[]`, the zeroth argument in `bash`, `$0`, is the name of the shell script being run. In the case of a function, this is `bash` itself.

To replicate the functions of the `dir.sh` script you wrote in Chapter 6, for example, you could add the same commands to a function definition, here called `listall`:

```
listall(){
  ls -la
  echo "Above are directory listings for this folder: "
  pwd
  date
}
```

You could of course define this function by typing or pasting these lines at a shell prompt—but as with aliases created in this manner, the effect will only last until that shell session is closed. To have functions available each time you open a new shell window, you must add their definitions to your `~/.bash_profile` (or `.profile` or `.bashrc` for Linux). In your `.bash_profile`, the definitions of these functions are written in the same manner as if they would be typed at the command line, so there is no need to include a `#!` line as you do with shell scripts.

All of the functions described in this section are available in the file `~/pcfb/examples/scripts/shellfunctions.sh`. Within shell functions, the `echo` statement is the equivalent of `print` in Python. This is used here in the `listall` function to print out a plain string of quoted text. The shell can also have variables, traditionally named with all capital letters. Remember in Chapter 6 how you set the value of the `PATH` variable:

```
export PATH="$PATH:$HOME/scripts"
```

In that case, you were setting a new value of `PATH` while reading the existing values of the variables `$PATH` and `$HOME`. In the shell, a variable name with a `$` is *read from*, while the same name without a `$` is *assigned* a value.

At the command line, try typing:

```
echo $HOME
```

This is the command-line way of saying "Print the value of the variable `HOME`." System-wide variables such as `HOME`, `PATH`, and `USER` are available within your shell functions. You can also use the special variables `$1`, `$2`, and `$@` to represent user arguments that follow your command. `$1` is a variable containing the first user-specified argument, `$2` is the second one (separated from the first argument by a space), and `$@` contains all the arguments together.

Create a simple function by typing these next lines in a shell window. Notice that you see a > at the prompt after having typed the first line; this indicates that the shell is waiting for you to finish your function definition with a close bracket:


```
repeater(){
 echo "$1 is what you said first"
 echo "$@ is everything you said"
}
```

With these lines, you have created a miniature shell script that is able to operate based on user input. Test out your function at the command line with some varied input:

```
host:~ lucy$ repeater 1 2 3
1 is what you said first
1 2 3 is everything you said
host:~ lucy$ repeater "1 2" 3
```

Notice that this works exactly like the `sys.argv[]` variable in Python, as described in Chapter 11.[5]

An immediate use for this would be if you had a frequently used shell command where the filename was followed by a long string of additional values. Typically, only the filename changes each time you use this command, but you can't easily use the ↑ key to edit the command line, because the change is not at the end.

For example, `phyml` is a program that infers maximum-likelihood trees from genetic sequence files. It takes the data filename first, followed by a long string of options:

```
phyml sequencesA.fta 1 i 1 100 WAG 0 8 e BIONJ y y
```

When using this command again on different datasets, you would specify the same options, but the sequence filename would change with each use.

For greater convenience and less typing, wrap the command within a function definition and replace the filename with `$1`:

```
myphyml(){
 echo "myphyml $1 performs 100 bootstraps on AA data"
 phyml $1 1 i 1 100 WAG 0 8 e BIONJ y y
}
```

[5] Similar to Python's `sys.argv[]`, the zeroth argument in `bash`, `$0`, is the name of the shell script being run. In the case of a function, this is `bash` itself.

Now you can invoke this function by typing just the following, and the filename (first parameter) will be inserted at the position occupied by `$1`:

```
myphyml sequencesA.fta
```

If your workflow requires other programs to be run on the output of `phyml`, you can add lines to your function to invoke those operations and accomplish all of the processing steps with a single command.

What if you want to optionally specify the number of bootstraps to run (currently occupied by the number `100`) as a second parameter in the function? You can use the shell's version of an `if` statement:

```
phymlaa(){
  BOOTS=100
  if [ $2 ]
  then
    BOOTS=$2
  fi
  phyml $1 1 i 1 $BOOTS WAG 0 8 e BIONJ y y
}
```

Notice that the end of the `if` statement is marked by `fi` (a backwards `if`). Indentation does not have special meaning in shell scripts, but do not use tabs in the body of the text if you are going to paste them into the command line—otherwise, the shell will try command completion just as if you were typing them. Spaces are critical within the lines of many shell commands: they must be omitted from assignments like `BOOTS=$2`, but must be present between the `[ ]` and `$2` or in any logical tests. The open square bracket `[` is actually a link to the test program, so to get help on logical tests, use the command `man test`. Comments in the shell language begin with #, just as they do in Python.

In this function, you start by setting a default value for the variable `BOOTS`. If the user only enters one parameter (the filename) then the default value of `100` is used. If they enter a second parameter `$2`, this is inserted into the command where the variable `$BOOTS` sits.

Notice the formulation of the `if` statement, where square brackets contain the logical expression. In this case, the test is just whether `$2` exists, to check for a second argument to the user input.

Even if you don't have the `phyml` program installed, you can test this function by replacing it with an `echo` command followed by the remainder of the line in quotes. Try different inputs on the command line to see how this affects the printing of the output lines. Using the `echo` command is a good way to test your scripts before actually deploying them with the power to overwrite other files.

We will now give some examples of shell function definitions, to give you a sense of what is possible.

Functions with user input

In some cases, you will want to pipe data from user arguments as input to a command in a function. To do this, you can `echo` the user input, add a pipe operator, and follow this with the name of the command. For example, if you have the program `blastall` installed locally on a server, and you wish to quickly do a BLAST search for a sequence that you have copied from a file, you can define this function:

```
myblastx(){
 echo "Blasting protein vs swissprot..."; date;
 echo $1 | blastall -d swissprot -p blastp -m 8 -i stdin
}
```

If you run this at the shell using the command:

```
myblastx "GKCPMSWAVLAPT"
```

then the `echo $1` statement will send the string to the `blastall` program as though it were read from a file.

A dictionary function

To look up words in the system dictionary, you can define a function to `grep` against the list:

```
function dict(){
   grep -Ei ${1} /usr/share/dict/words
}
```

Run this command using word fragments or `grep` wildcards:

```
dict kkee
dict noi | grep  ion
dict ^ct
dict p.z.z
```

Translating characters

To do a batch conversion of end-of-line characters from `\r` (carriage return) to `\n` (linefeed) as required by many Unix scripts, you can use the `tr` function, which translates one character to another in the data stream. In some of these examples, the bracket characters `{}` are placed around variable names, as in `${1}`. This helps the script handle cases where the variable includes spaces:

```
unix2mac(){
# this line tests if the number of arguments is zero
if [ $# -lt 1 ]
then
    echo "convert mac to unix end of line"
else
    tr '\r' '\n' < "${1}" > u_"${1}"
    echo "converting ${1} to u_${1}"
fi
}
```

Looping through all arguments passed to a function

Shell functions can also include `for` loops. These operate on a series of space-separated values, the same way that `$1` and `$2` access consecutive values separated by spaces at the prompt. To loop through all the arguments given at the command line, you can use the basic syntax of:

```
for ITEM in $@; do
```

The list of items can also be anything that would be correctly interpreted at the command line. So to loop through all the text files in a directory, you can say:

```
for FILE in *.txt; do
  echo $FILE
done
```

In bash commands and functions, a semicolon ends the command in the same way that pressing return would. You can use this to join several commands onto a single line. The three-line script above can be rephrased as follows:

```
for FILE in *.txt; do echo $FILE; done
```

In fact, if you use ↑ to step back through your `bash` history, that is how the three-line operation will be presented. Adding other statements after the `echo` statement separated by semicolons can insert those commands into the loop.

The following function loops through a group of filenames indicated at the command line (whether typed individually or signified by `*.txt`) and renames them from `filename.txt` to `u_filename.dat`. Step through the example shown here to see how each file in succession is processed:

```
renamer(){
 # Edit the prefix and extension to change how this works
 EXT="dat"
 PRE="u_"
```



```
# test if there is one or more file name provided
if [ $# -lt 1 ]
then
  echo "Rename a file.txt list as $PREfile.$EXT"
else
  for FILENAME in "$@"
    do
      ROOTNAME="${FILENAME%.*}"

      cp "$FILENAME" "$PRE$ROOTNAME.$EXT"
    echo "Copying $FILENAME to $PRE$ROOTNAME.$EXT"
    done
 fi
}
```

In a shell function, the `$@` represents the list of all arguments sent to the script. If you type `*.txt` at the command line, then `$@` will be a list of all the matching filenames. The loop cycles through each parameter contained in the master argument `$@`. For each filename, the program strips off the extension using the unusual shell expression `${FILENAME%.*}` The percent sign starts a search term which gives everything up to the last period and deletes what comes after it. Remember that `*` is the most inclusive wildcard in the shell, just as `.*` is in regular expressions. Here, the period is interpreted literally as that character, not as a wildcard. An equivalent operation acting on a full directory name, in which each element is separated by a slash, would be `${RESULT%/*}`. This gives everything up to, but not including, the last slash. This is useful for processing directory names to find the enclosing folder.

This function uses the "root name" of the file as the basis for constructing a new filename by joining variables with other bits of text (`$PRE$ROOTNAME.$EXT` includes three variables and a plain text period).

Removing file extensions

Another use of the extension-removing `${VAR%/*}` syntax is to find the name of a folder, given the full path to a file. For example, you might want to change into a directory containing a particular program—say, for example, where the `grep` program is stored. Normally, you would have to type `which grep` to find the name of the directory, and then `cd` and retype or paste the part of the path preceding the filename:

```
lucy$ which grep
/usr/bin/grep
lucy$ cd /usr/bin
host:bin lucy$
```

A quick shell function can accomplish these steps in one command: first, finding the location of a command, and then cd'ing to the path with the name of the program removed from the end:

```
whichcd(){
   RESULT=`which ${1}`
   cd ${RESULT%/*}
}
```

This function makes use of yet another shell convention: the backtick symbol ` (usually located near the esc key). This shows up in other programming languages as well, and it essentially represents the text that is output when performing a command in a shell window. In this case, if you type the command `whichcd zip`, it executes a `which` command on the `zip` program, and assigns the value `/usr/bin/zip` to the variable named `RESULT`. It then strips the program name off the end of the directory name and uses that for the `cd` command.

MAKING SURE TO ESCAPE Some function definitions will work when pasted at the command line, but they won't when loaded from your `.bash_profile`. The problem here is typically that characters like `&` or `$` need to be escaped with `\` when read from a file. If you are using characters like this in a function without success, try both escaping them and not escaping them, to see which way works.

Experiment with the `${%.*}` at the command line by setting up a variable and then using `echo` to examine the output:

```
lucy$ TEST="/MyDir/MyFile.txt"
lucy$ echo ${TEST%.*}
/MyDir/MyFile
lucy$ echo ${TEST%/*}
/MyDir
```

Finding files

The `find` command is a way to search your computer for files that match certain criteria. Its syntax, though, is relatively confusing. To create a shortcut for one of its uses—searching all nested folders relative to the current location—you can define this function:

```
findf(){
  find \. -name ${1} -print
}
```


In this shortcut, the command will start at the current location (`.`) and use the first argument `$1` as the name to search for. This provides a more intuitive interface for locating files with the `find` command.

Revisiting piped commands

Earlier in the chapter you saw how to extract a count of amino acids from a PDB-format protein structure file. To create a function from this command, simply replace the name of the file with `$1` and insert it into a function definition. As in Python, you can use a backslash to escape out the end-of-line character and split the single long line into two lines:

```
countpdb(){
 grep ATOM $1 | grep -v REMARK | cut -c 18-21,24-26 |  sort | uniq \
 | cut -f 1 -d " " | uniq -c | sort -nr
}
```

A more generic command to enumerate the unique items in a particular column of a tab-delimited table can be defined using:

```
countlist(){
 echo "### Type countlist followed by file name, then column number"
 echo "### Returns a sorted list of the unique items in that column"
 cut -f ${2} ${1}| sort | uniq -c | sort -n -r
}
```

Repeating operations with loops

Imagine you want to test the effects of changing a single parameter on an analysis, but that running through even a single operation takes an hour. Instead of checking on your computer every hour and relaunching the command, or generating a script with all of the commands listed in succession, you can use a shell function to loop through the series of parameters. The basic syntax of a shell `for` loop is:

```
for k in {1..10}; do
 echo $k
done
```

Note the brackets followed by a semicolon and the word `do`, to start the loop, then the word `done` to close the loop.

Now try two methods of creating numeric loops in a shell function. The first increases the value of the parameter `x` from `20` to `40`, and saves the output of

the ABYSS program in files named `contigs_20.fa`, `contigs_21.fa`, on up to `contigs_40.fa`:

```
for x in {20..40}; do
ABYSS -k$x reads.fa -o contigs_$x.fa
done
```

The second varies x from 30 to 45 by 5:

```
date;
for ((x=30; x<=45; x+=5)); do
   ABYSS -k$x reads.fta -o contigs_reads-k$x.fa;
   date;
done
```

Wrappers

A **wrapper** is a program that controls and expands the functionality of another program. At its core, the wrapper is calling the other program from the command line; however, the wrapper can take care of a variety of other tasks as well, such as reformatting input data, constructing complex strings of arguments (including creating fields that must be calculated), and parsing program output into a more convenient format. Wrappers can make using an existing program a bit more convenient but they can also enable entirely new analyses.

A combination of Python and shell commands can be an excellent way to build a wrapper. Even MATLAB programs can be executed in command-line mode, so they can be included in your automated scripts. The `exifparse.py` script shown here uses the `exiftool` shell program described in Chapter 19 to extract scale bar information that has been embedded in a series of electron microscope images; it also uses Python's `os.popen().read()` function to run a shell command and capture its output:

```
#! /usr/bin/env python
""" Generate a table of pixels per micron for TEM
images, using exiftool. Run from a folder containing
images or subfolders with images...

Requires the program exiftool to be installed and in the path:
http://www.sno.phy.queensu.ca/~phil/exiftool/"""

import os
import sys
```



```
DirList = os.popen('ls -F| grep \/','r').read().split()
DirList.append('./')
#print DirList

for Direct in DirList:
   sys.stderr.write("Directory "+Direct+'----------------\n')
   FileList=os.popen('ls ' + Direct +" |grep tif").read().split()

"""Exif data are in the format:
Image Description : AMT Camera System.11/18/09.14:12.8000.7.0.80.1.Imaging...
-79.811.-552.583..XpixCal=65.569.YpixCal=65.569.Unit=micron.##fv3
"""
   if len(FileList)==0:
      sys.stderr.write("No files found in "+Direct+'\n')
   else:
#     print "Found", FileList ←Commented out
      for Path in FileList:
         # The statement below runs a bash command and captures the output
         ExifData=os.popen('exiftool '+Direct+Path +' | grep YpixCal').read()
         SecondHalf=ExifData.split('XpixCal=')[1].strip()
         NumberOnly=SecondHalf.split('.YpixCal')[0].strip()
         print Path +"\t"+NumberOnly
```

Thoughts on pipelines

Creating an automated workflow that stitches several programs together, or just does all the dirty work involved in performing the same operation repeatedly, can save a great deal of time. However, there are other benefits to automation that may be even more important in some cases. For one, you know that the task is being performed consistently without deviations in operator input from analysis to analysis. Your automation of a task also serves as a record of what you did. When it comes time to write up your analyses you have a record of how the programs were called, preserved in the very script that called them. This kind of detail is often difficult to document—especially since most biologists are better at keeping notebooks about how they collected their data than they are at recording exactly how the dataset was analyzed.

There is no single solution for automating every kind of task. Some programs are designed so that it is easy for other programs to talk to them directly.[6] Other programs are not designed to interact with other software, but still feature capabilities within the graphical interface that make automation possible. Unfortunately, some programs have user interfaces that make automation difficult or impossible; worst are those with a graphical user interface but no command-line interface, and which don't read external configuration files or have any built-in scripting abilities.

[6] When an interface is provided for a program specifically to ease direct communication with other software, this is called an API, or application programming interface.

In general, almost any command-line program can be tricked by another program into thinking that it is interacting with a person. Armed with your knowledge of shell and programming operations, you should be able to accomplish more with less effort.

SUMMARY

You have learned how to:

- Show just `head` or `tail` of a file
- Use `cut` to extract columns of text
- `sort` lines in a file
- Find and count the unique lines with `uniq`
- Define one-line aliases using `alias aliasname="alias command"`
- Create more involved shell functions using `functionname(){}`
- Create loops in the shell
- Create a wrapper that automates and facilitates the execution of an existing program

Recommended reading

"Bash shell scripting tutorial," http://steve-parker.org/sh/sh.shtml.

Taylor, Dave. *Wicked Cool Shell Scripts: 101 Scripts for Linux, Mac OS X, and Unix Systems.* San Francisco: No Starch Press, 2004.