CHAPTER 1

Unix

1.1 What Is Unix?

Unix is an operating system, which means that it is the software that lets you interface with the computer. It was developed in the 1970s by a group of programmers at the AT&T Bell laboratories. The new operating system was an immediate success in academic circles, with many scientists writing new programs to extend its features. This mix of commercial and academic interest led to the many variants of Unix available today (e.g., OpenBSD, Sun Solaris, Apple's OS X), collectively denoted as *nix systems. Linux is the open source Unix clone whose "engine" (kernel) was written from scratch by Linus Torvalds with the assistance of a loosely knit team of hackers from across the internet. Ubuntu is a popular Linux distribution (version of the operating system).

All *nix systems are multiuser, network-oriented, and store data as plain text files that can be exchanged between interconnected computer systems. Another characteristic is the use of a strictly hierarchical file system, discussed in section 1.3.2.

This chapter focuses primarily on the use of the Unix shell. The shell is the interface that is used to communicate with the core of the operating system (kernel). It processes the commands you type, translates them for the kernel, and shows you the results of your operations. The shell is often run within an application called the *terminal*. Together, the shell and terminal are also referred to as a *command-line interface* (CLI), an interface that allows you to input commands as successive lines of text. Though technically not correct, the terms shell, command line (interface), and terminal are often used interchangeably. Even if you have never worked with a command-line interface, you have surely seen one in a movie: Hollywood likes the stereotype of a hacker typing code in a small window with a black background (i.e., command-line interface).

Today, several shells are available, and here we concentrate on the most popular one, the Bash shell, which is the default shell in Ubuntu and OS X. When working on the material presented in this book, it is convenient, though not strictly necessary, to work in a *nix environment. Git Bash for Windows emulates a Unix shell. As the name Git Bash implies, it also uses the Bash shell.

1.2 Why Use Unix and the Shell?

Many biologists are not familiar with using *nix systems and the shell, but rather prefer graphical user interfaces (GUIs). In a GUI, you work by interacting with graphical elements, such as buttons and windows, rather than typing commands, as in a command-line interface. While there are many advantages to working with GUIs, working in your terminal will allow you to automate much of your work, scale up your analysis by performing the same tasks on batches of files, and seamlessly integrate different programs into a well-structured pipeline.

This chapter is meant to motivate you to get familiar with command-line interfaces, and to ease the initially steep learning curve. By working through this chapter, you will add a tool to your toolbox that is the foundation of many others—you might be surprised to find out how natural it will become to turn to your terminal in the future. Here are some more reasons why learning to use the shell is well worth your effort:

First, Unix is an operating system written by programmers for programmers. This means that it is an ideal environment for developing your code and managing your data.

Second, hundreds of small programs are available to perform simple tasks. These small programs can be strung together efficiently so that a single line of Unix commands can perform complex operations, which otherwise would require writing a long and complicated program. The ability to create these pipelines for data analysis is especially important for biologists, as modern research groups produce large and complex data sets whose analysis requires a level of automation and reproducibility that would be hard to achieve otherwise. For instance, imagine working with millions of files by having to open each one of them manually to perform an identical task, or try opening your single 80 Gb whole-genome sequencing file in software with a GUI! In Unix, you can string a number of small programs together, each performing a simple task, and create a complex pipeline that can be stored in a script (a text file containing all the commands). Such a script makes your work 100% reproducible. Would you be able to repeat the exact series of 100 clicks of a complex analysis in a GUI? With a script, you will always obtain the same

result! Furthermore, you will also save much time. While it may take a while to set up your scripts, once they are in place, you can let the computer analyze all of your data while you're having a cup of coffee. This level of automation is what we are striving for throughout the book, and the shell is the centerpiece of our approach.

Third, text is the rule: If your data are stored in a text file, they can be read and written by any machine, and without the need for sophisticated (and expensive) proprietary software. Text files are (and always will be) supported by any operating system and you will still be able to access your data decades from today (while this is not the case for most proprietary file formats). The text-based nature of Unix might seem unusual at first, especially if you are used to graphical interfaces and proprietary software. However, remember that Unix has been around since the early 1970s and will likely be around at the end of your career. Thus, the hard work you put into learning Unix will pay off over a lifetime.

The long history of Unix means that a large body of tutorials and support websites are readily available online. Last but not least, Unix is very stable, robust, secure, and—in the case of Linux—freely available.

In the end, it is almost impossible for a professional scientist to entirely avoid working in a Unix shell: the majority of high-performance computing platforms (computer clusters, large workstations, etc.) run a Unix or Linux operating system. Similarly, the transfer of large files, websites, and data between machines is often accomplished through command-line interfaces.

Mastering the skills presented in this chapter will allow you to work with large files (or with many files) effortlessly. Most operations can be accomplished without the need to open the file(s) in an editor, and can be automated very easily.

1.3 Getting Started with Unix

1.3.1 Installation

The Linux distribution Ubuntu and Apple's OS X are members of the *nix family of operating systems. If you are using either of them, you do not need to install any specific software to follow the material in this chapter.

Microsoft Windows is not based on a *nix system; you can, however, recreate a Unix environment within Windows by installing the Ubuntu operating system in a virtual machine. Alternatively, Windows users can install Git Bash, a clone of the Bash terminal. It provides basic Unix and Git commands and many other standard Unix functionalities can be installed.

Please find instructions for its installation in CSB/unix/installation. Windows also ships with the program Command Prompt, a command-line interface for Windows. However, many commands differ from their Bash shell counterparts, so we will not cover these here.

1.3.2 Directory Structure

In Unix we speak of "directories," while in a graphical environment the term "folder" is more common. These two terms are interchangeable and refer to a structure that may contain subdirectories and files. The Unix directory structure is organized hierarchically in a tree. Figure 1.1 illustrates a directory structure in the OS X operating system. The topmost directory in the hierarchy is also called the "root" directory and is denoted by an individual slash (/). The precise architecture varies among the different operating systems, but there are some important directories that branch off the root directory in most operating systems:

`/bin` Contains several basic programs
`/dev` Contains the files connecting to devices such as the keyboard, mouse, and screen
`/etc` Contains configuration files
`/tmp` Contains temporary files

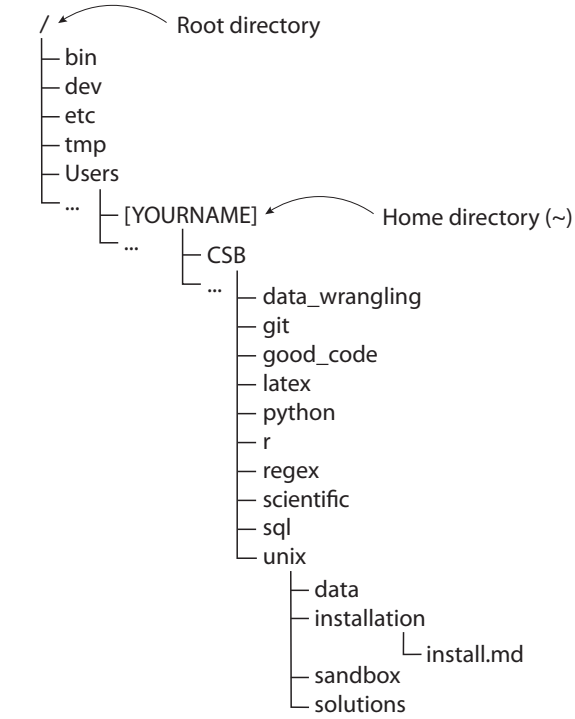
Another important directory is your home directory (also called the login directory), which is the starting directory when you open a new shell. It contains your personal files, directories, and programs. The tilde (~) symbol is shorthand for the home directory in Ubuntu and OS X. The exact path to your home directory varies slightly among different operating systems. To print its location, open a terminal and type[1]

```
echo $HOME
```

The command `echo` prints a string to the screen. The dollar sign indicates a variable. You will learn more about variables in section 1.7.

If you followed the instructions in section 0.5, you should have created a directory called `CSB` in your home directory. In Ubuntu the location is `/home/YOURNAME/CSB`, in OS X it is `/Users/YOURNAME/CSB`. Windows users need

1. Windows users, use Git Bash or type `echo %USERPROFILE%` at the Windows Command Prompt.



Full path of the file install.md:
`/Users/[YOURNAME]/CSB/unix/installation/install.md`

Figure 1.1. An example of the directory structure in the OS X operating system. It shows several directories branching off the root directory (/). In OS X the home directory (~) is a subdirectory of `Users`, and in UNIX it is a subdirectory of `home`. If you have followed the instructions in section 0.5, you will find the directory `CSB` in your home directory. As an example, we show the full path of the file `install.md`.

to decide where to store the directory. Within `CSB`, you will find several directories, one for each chapter (e.g., `CSB/unix`). Each of these directories contains the following subdirectories:

`installation` The instructions for installing the software needed for the chapter are contained here. These are also available online.[2]

`sandbox` This is the directory where we work and experiment.

`data` This directory provides all data for the examples and exercises, along with the corresponding citations for papers and websites.

2. github.com/CSB-book/CSB.

`solutions` The detailed solutions for the exercises are here, as well as sketches of the solutions in plain English (*pseudocode*) that you should consult if you don't know how to proceed with an exercise. Solutions for the "Intermezzo" sections are available at the end of the book.

When you navigate the system, you are in one directory and can move deeper in the tree, or upward toward the root. Section 1.4.3 discusses the commands you can use to move between the hierarchical levels and determine your location within the directory structure.

1.4 Getting Started with the Shell

In Ubuntu, you can open a shell by pressing Ctrl+Alt+T, or by opening the dash (hold the Meta key) and typing `Terminal`. In OS X, you want to open the application `Terminal.app`, which is located in the folder `Utilities` within `Applications`. Alternatively, you can type "Terminal" in `Spotlight`. Windows users can launch Git Bash or another terminal emulator. In all systems, the shell automatically starts in your home directory.

When you open a terminal, you should see a line (potentially containing information on your user name and location), ending with a dollar ($) sign. When you see the dollar sign, the terminal is ready to accept your commands. Give it a try and type

```
# display date and time
$ date
```

In this book, a $ sign at the beginning of a line of code signals that the command has to be executed in your terminal. You do not need to type the $ sign in your terminal, only copy the command that follows it. A line starting with a hash symbol (#) means that everything that follows is a comment. While Unix ignores comments, you will find hints, reminders, and explanations there. Make plenty of use of comments to document your own code. When writing multiple lines of comments, start each with #.

In Unix, you can use the Tab key to reduce the amount you have to type, which in turn reduces the probability of making mistakes. When you press Tab in a (properly configured) shell, it will try to automatically complete your command, directory, or file name. If multiple completions are possible, you can display them all by hitting the Tab key twice. Additionally, you

can navigate the history of commands you have typed by using the up/down arrows. This is very convenient as you do not need to retype a command that you have executed recently. The following box lists keyboard shortcuts that help pace through long lines of code.

Ctrl+A	Go to the beginning of the line.
Ctrl+E	Go to the end of the line.
Ctrl+L	Clear the screen.
Ctrl+U	Clear the line before the cursor position.
Ctrl+K	Clear the line after the cursor.
Ctrl+C	Kill the command that is currently running.
Ctrl+D	Exit the current shell.
Alt+F	Move cursor forward one word (in OS X, Esc+F).
Alt+B	Move cursor backward one word (in OS X, Esc+B).

Mastering these and other keyboard shortcuts will save you a lot of time. You may want to print this list (available at computingskillsforbiologists.com/terminalshortcuts) and keep it next to your keyboard—when used consistently you will have them all memorized and will start using them automatically.

1.4.1 Invoking and Controlling Basic Unix Commands

Some commands can be executed simply by typing their name:

```
# print a simple calendar
$ cal
```

However, you can pass an *argument* to the command to alter its behavior:

```
# pass argument to cal to print specific year
$ cal 2020
```

Some commands may require obligatory arguments and will return an error message if they are missing. For example, the command to copy a file needs two arguments: what file to copy, and where to copy it (see section 1.5.1).

In addition, all commands can be modified using *options* that are specific to each command. For instance, we can print the calendar in Julian format, which labels each day with a number starting on January 1:

```
# use option -j to display Julian calendar
$ cal -j
```

Options can be written using either a dash followed by a single letter (older style, e.g., `-j`) or two dashes followed by words (newer style, e.g., `--julian`). Note that not every command offers both styles.

In Unix, the placement of spaces between a command and its options or arguments is important. There needs to be a space between the command and its options, and between multiple arguments. However, if you are supplying multiple options to a command you can string them together (e.g., `-xvzf`).

If you are using Unix commands for the first time, it might seem odd that you usually do not get a message or response after executing a command (unless the command itself prints to the screen). Some commands provide feedback on their activity when you request it using an option. Otherwise, seeing no (error) message means that your command worked.

Last but not least, it is important to know how to interrupt the execution of a command. Press Ctrl+C to halt the command that is currently running in your shell.

1.4.2 How to Get Help in Unix

Unix ships with hundreds of commands. As such, it is impossible to remember them all, let alone all their possible options. Fortunately, each command is described in detail in its manual page, which OS X and Ubuntu users can access directly from the shell by typing `man [COMMAND]`. Use arrows to scroll up and down and press q to close the manual page. Users of Git Bash can search online for `unix man page [COMMAND]` to find many sites displaying the manual of Unix commands.

Checking the exact behavior of a command is especially important, given that the shell will execute any command you type without asking whether you know what you're doing (so that it will promptly remove all of your files,

if that's the command you typed). You may be used to more forgiving (and slightly patronizing) operating systems in which a pop-up window will warn you whenever something you're doing is considered dangerous. However, don't feel afraid to use the shell as much as possible. The really dangerous commands are all very specific—there is very little chance that you will destroy something accidentally simply by hitting the wrong key.

1.4.3 Navigating the Directory System

You can navigate the hierarchical Unix directory system using the following commands:

`cd` Change directory. The command requires one argument: the path to the directory you want to change into. There are a few options for the command that speed up navigation through the directory structure:

`cd ..` Move one directory up.
`cd /` Move to the root directory.
`cd ~` Move to your home directory.
`cd -` Go back to the directory you visited previously (like "Back" in a browser).

```
# assuming you saved CSB in your home directory
# navigate to the sandbox in the CSB/unix directory
cd ~/CSB/unix/sandbox
```

`pwd` Print the path of the current working directory. This command prints your current location within the directory structure.

```
$ pwd
/Users/mwilmes/CSB/unix/sandbox
# this may look different depending on your system
```

`ls` List the files and subdirectories in the current directory. There are several useful options:

`ls -a` List all files (including hidden files).

`ls -l` Return a long list with details on access permissions (see section 1.6.7), the number of links to a file, the user and group owning it, its size, the time and date it was changed last, and its name.

`ls -lh` Display file sizes in human readable units (K, M, G for kilobytes, megabytes, gigabytes, respectively).

One can navigate through the directory hierarchy by providing either the *absolute* path or a *relative* path. An example of the absolute path of a directory is indicated at the bottom of figure 1.1. The full path of the file `install.md` is indicated, starting at the root. A relative path is defined with respect to the current directory (use `pwd` to display the absolute path of your current working directory). Let's look at an example:

```
# absolute path to the directory CSB/python/data
# if CSB is in your home directory
$ cd ~/CSB/python/data
# relative path to navigate to CSB/unix/data
# remember: the Tab key provides autocomplete
$ cd ../../unix/data
# go back to previous directory (CSB/python/data)
$ cd -
```

You can always use either the absolute path or a relative path to specify a directory. When you navigate just a few levels higher or deeper within the tree, a relative path usually means less to type. If you want to jump somewhere far within the tree, the absolute path might be the better choice.

Note that directory names in a path are separated with a forward slash (/) in Unix but usually with a backslash (\) in Windows (e.g., when you look at the path of a file using File Explorer). However, given that Git Bash emulates a Unix environment, you will find it uses a forward slash despite working with the Windows operating system.

In Unix, a full path name cannot have any spaces. Spaces in your file or directory names need to be preceded by a backslash (\). For example, the file "`My Manuscript.txt`" in the directory "`Papers and reviews`" becomes `Papers\ and\ reviews/My\ Manuscript.txt`. To avoid such unruly path names, an underscore (_) is recommended for separating elements in the names of files and directories, rather than a space. If you need to refer to an existing file or directory that has spaces in its name, use quotation marks around it.

```
# does not work
cd Papers and reviews
# works but not optimal
cd Papers\ and\ reviews
cd "Papers and reviews"
# when creating files or directories use
# underscores to separate elements in their names
cd Papers_and_reviews
```

Intermezzo 1.1

(a) Go to your home directory.
(b) Navigate to the `sandbox` directory within the `CSB/unix` directory.
(c) Use a relative path to go to the `data` directory within the `python` directory.
(d) Use an absolute path to go to the `sandbox` directory within `python`.
(e) Return to the `data` directory within the `python` directory.

1.5 Basic Unix Commands

1.5.1 Handling Directories and Files

Creating, manipulating and deleting files or directories is one of the most common tasks you will perform. Here are a few useful commands:

`cp` Copy a file or directory. The command requires two arguments: the first argument is the file or directory that you want to copy, and the second is the location to which you want to copy. In order to copy a directory, you need to add the option `-r` which makes the command *recursive*. The directory and its contents, including subdirectories (and their contents) will be copied.

```
# copy a file from unix/data directory into sandbox
# if you specify the full path,
# your current location does not matter
$ cp ~/CSB/unix/data/Buzzard2015_about.txt ~/CSB/unix/
    ↳ sandbox/
# assuming your current location is the unix sandbox,
```

```
# we can use a relative path
$ cp ../data/Buzzard2015_about.txt .
# the dot is shorthand to say "here"
# rename the file in the copying process
cp ../data/Buzzard2015_about.txt ./Buzzard2015_about2.txt
# copy a directory (including all subdirectories)
cp -r ../data .
```

`mv` Move or rename a file or directory. You can move a file by specifying two arguments: the name of the file or directory you want to move, and the destination. You can also use the `mv` command to rename a file or directory. Simply specify the old and the new file name in the same location.

```
# move the file to the data directory
$ mv Buzzard2015_about2.txt ../data/
# rename a file
$ mv ../data/Buzzard2015_about2.txt ../data/
    ↳ Buzzard2015_about_new.txt
# easily manipulate a file that is
# not in your current working directory
```

`touch` Update the date of last access to the file. Interestingly, if the file does not exist, this command will create an empty file.

```
# inspect the current contents of the directory
$ ls -l
# create a new file (you can list multiple files)
$ touch new_file.txt
# inspect the contents of the directory again
$ ls -l
# if you touch the file a second time,
# the time of last access will change
```

`rm` Remove a file. It has some useful options: `rm -r` deletes the contents of a directory recursively (i.e., including all files and subdirectories in it). Use this command with caution or in conjunction with the `-i` option, which prompts the user to confirm the action. The option `-f` forcefully removes a write-protected file (such as a directory under version control) without a prompt.

Again, use with caution as there is no trash bin that allows you to undo the removal!

```
$ rm -i new_file.txt
remove new_file.txt? y
# confirm deletion with y (yes) or n (no)
```

`mkdir` Make a directory. To create nested directories, use the option `-p`:

```
$ mkdir -p d1/d2/d3
# remove the directory by using command rm recursively
$ rm -r d1
```

1.5.2 Viewing and Processing Text Files

Unix was especially designed to handle text files, which is apparent when considering the multitude of commands dealing with text. Here are a few popular ones with a selection of useful options:[3]

`less` Progressively print a file to the screen. With this command you can instantly take a look at very large files without the need to open them. In fact, `less` does not load the entire file, but only what needs to be displayed—making it much faster than a text editor. Once you enter the `less` environment, you have many options to navigate through the file or search for specific patterns. The simplest are Ctrl+F to jump one screen forward and Ctrl+B to jump one back. See more options by pressing h or have a look at the manual page. Pressing `q` quits the `less` environment.[4]

```
# assuming you're in CSB/unix/data
$ less Marra2014_data.fasta
>contig00001  length=527  numreads=2  gene=isogroup00001
    ↳ status=it_thresh
```

3. We recommend skimming the manual pages of each command to get a sense of their full capabilities.

4. Funny fact: there is a command called `more` that does the same thing, but with less flexibility. Clearly, in Unix, `less` is `more`.

```
ATCCTAGCTACTCTGGAGACTGAGGATTGAAGTTCAAAGTCAGCTCAAGCAAGAGATTT
...
```

`cat` Concatenate and print files. The command requires at least one file name as argument. If you provide only one, it will simply print the entire file contents to the screen. Providing several files will concatenate the contents of all files and print them to the screen.

```
# concatenate files and print to screen
$ cat Marra2014_about.txt Gesquiere2011_about.txt
    ↳ Buzzard2015_about.txt
```

`wc` Line, word, and byte (character) count of a file. The option `-l` returns the line count only and is a quick way to get an idea of the size of a text file.

```
# count lines, words, and characters
$ wc Gesquiere2011_about.txt
       8      64     447 Gesquiere2011_about.txt
# count lines only
$ wc -l Marra2014_about.txt
      14 Marra2014_about.txt
```

`sort` Sort the lines of a file and print the result to the screen. Use option `-n` for numerical sorting and `-r` to reverse the order. The option `-k` is useful to sort a delimiter-separated file by a specific column (more on this command in section 1.6.3).

```
# print the sorted lines of a file
$ sort Gesquiere2011_data.csv
100     102.56  163.06
100     117.05  158.01
100     133.4   94.78
...
# sort numerically
$ sort -n Gesquiere2011_data.csv
```

```
maleID  GC      T
1       32.65   59.94
1       51.09   35.57
1       52.72   43.98
...
```

`uniq` Show only the unique lines of a file. The contents need to be sorted first for this to work properly. Section 1.6.2 describes how to combine commands. The option `-c` returns a count of occurrence for each unique element in the input.

`file` Determine the type of a file. Useful to identify Windows-style line terminators[5] before opening a file.

```
$ file Marra2014_about.txt
Marra2014_about.txt: ASCII English text
```

`head` Print the head (i.e., first few lines) of a file. The option `-n` determines the number of lines to print.

```
# display first two lines of a file
head -n 2 Gesquiere2011_data.csv
maleID  GC      T
1       66.9    64.57
```

`tail` Print the tail (i.e., last few lines) of a file. The option `-n` controls the number of lines to print (starting from the end of the file). The option can also be used to display everything but the first few lines.

```
# display last two lines of file
$ tail -n 2 Gesquiere2011_data.csv
127     108.08  152.61
127     114.09  151.07
```

5. Covered in section 1.9.2.

```
# display from line 2 onward
# (i.e., removing the header of the file)
$ tail -n +2 Gesquiere2011_data.csv
1       66.9    64.57
1       51.09   35.57
1       65.89   114.28
...
```

`diff` Show the differences between two files.

Intermezzo 1.2

To familiarize yourself with these basic Unix commands, try the following:

(a) Go to the `data` directory within `CSB/unix`.
(b) How many lines are in file `Marra2014_data.fasta`?
(c) Create the empty file `toremove.txt` in the `CSB/unix/sandbox` directory without leaving the current directory.
(d) List the contents of the directory `unix/sandbox`.
(e) Remove the file `toremove.txt`.

1.6 Advanced Unix Commands

1.6.1 Redirection and Pipes

So far, we have printed the output of each command (e.g., `ls`) directly to the screen. However, it is easy to *redirect* the output to a file or to *pipe* the output of one command as the input to another command. Stringing commands together using pipes is the real power of Unix, letting you perform complex tasks on large amounts of data using a single line of commands.

First, we show how to redirect the output of a command into a file:

```
$ [COMMAND] > filename
```

Note that if the file `filename` exists, it will be overwritten. If instead we want to append the output of a command to an existing file, we can use the >> symbol, as in the following line:

```
$ [COMMAND] >> filename
```

When the command is very long and complex, we might want to redirect the contents of a file as input to a command, "reversing" the flow:

```
$ [COMMAND] < filename
```

To run a few examples, let's start by moving to our `sandbox`:

```
$ cd ~/CSB/unix/sandbox
```

The command `echo` can be used to print a string on the screen. Instead of printing to the screen, we redirect the output to a file, effectively creating a file containing the string we want to print:

```
$ echo "My first line" > test.txt
```

We can see the result of our operation by printing the file to the screen using the command `cat`:

```
$ cat test.txt
```

To append a second line to the file, we use `>>`:

```
$ echo "My second line" >> test.txt
$ cat test.txt
```

We can redirect the output of any command to a file.

Here is an example: Your collaborator or laboratory machine provided you with a large number of data files. Before analyzing the data, you want to get a sense of how many files need to be processed. If there are thousands of files, you wouldn't want to count them manually or even open a file browser that could do the counting for you. It is much simpler and faster to type a few Unix commands.

We will use `unix/data/Saavedra2013` as an example of a directory with many files. First, we create a file that lists all the files contained in the directory:

```
# current directory is the unix sandbox
# create a file listing the contents of a directory
$ ls ../data/Saavedra2013 > filelist.txt
# look at the file
$ cat filelist.txt
```

Now we want to count how many lines are in the file. We can do so by calling the command `wc -l`:[6]

```
# count lines in a file
$ wc -l filelist.txt
# remove the file
$ rm filelist.txt
```

However, we can skip the creation of the intermediate file (`filelist.txt`) by creating a short pipeline. The pipe symbol (`|`) tells the shell to take the output on the left of the pipe and use it as the input for the command on the right of the pipe. To take the output of the command `ls` and use it as the input of the command `wc` we can write

```
# count number of files in a directory
$ ls  ../data/Saavedra2013 | wc -l
```

We have created our first, simple pipeline. In the following sections, we are going to build increasingly long and complex pipelines. The idea is always to start with a command and progressively add one piece after another to the pipeline, each time checking that the result is the desired one.

1.6.2 Selecting Columns Using `cut`

When dealing with tabular data, you will often encounter the comma-separated values (CSV) standard file format. As the name implies, the data are usually structured by commas, but you may find CSV files using other

6. This is the lowercase letter `L` as in line.

delimiters such as semicolons or tabs (e.g., because the data values contain commas and spaces). The CSV format is text based and platform and software independent, making it the standard output format for many experimental devices. The versatility of the file format should also make it your preferred choice when manually entering and storing data.[7] Most of the exercises in this book use the CSV file format in order to highlight how easy it is to read and write these files using different programming languages.

The main Unix command you want to master for comma-, space-, tab-, or character-delimited text files is `cut`. To showcase its features, we work with data on the generation time of mammals published by Pacifici et al. (2013). First, let's make sure we are in the right directory (`~/CSB/unix/data`). Then, we can print the header (the first line, specifying the contents of each column) of the CSV file using the command `head`, which prints the first few lines of a file on the screen, with the option `-n 1`, specifying that we want to output only the first line:

```
# change directory
$ cd ~/CSB/unix/data
# display first line of file (i.e., header of CSV file)
$ head -n 1 Pacifici2013_data.csv
TaxID;Order;Family;Genus;Scientific_name;...
```

We can see that the data are separated by semicolons. We pipe the first line of the file to `cut` and use the option `-d ";"` to specify the delimiter. The additional option `-f` lets us extract specific columns: here column 1 (`-f 1`), or the first four columns (`-f 1-4`).

```
# take first line, select 1st column of ";"-separated file
$ head -n 1 Pacifici2013_data.csv | cut -d ";" -f 1
TaxID

$ head -n 1 Pacifici2013_data.csv | cut -d ";" -f 1-4
TaxID;Order;Family;Genus
```

Remember to use the Tab key to autocomplete file names and the arrow keys to access your command history.

7. If you need to store and process large data sets, you should consider databases, which we explore in chapter 10, as an alternative.

In the next example we work with the contents of our data file. We specify a delimiter, extract specific columns, and pipe the result to the `head` command in order to display only the first few elements:

```
# select 2nd column, display first 5 elements
$ cut -d ";" -f 2 Pacifici2013_data.csv  | head -n 5
Order
Rodentia
Rodentia
Rodentia
Macroscelidea

# select 2nd and 8th columns, display first 3 elements
$ cut -d ";" -f 2,8 Pacifici2013_data.csv  | head -n 3
Order;Max_longevity_d
Rodentia;292
Rodentia;456.25
```

Now, we specify the delimiter, extract the second column, skip the first line (the header) using the `tail -n +2` command (i.e., return the whole file starting from the second line), and finally display the first five entries:

```
# select 2nd column without header, show 5 first elements
$ cut -d ";" -f 2 Pacifici2013_data.csv  | tail -n +2 |
    ↳ head -n 5
Rodentia
Rodentia
Rodentia
Macroscelidea
Rodentia
```

We pipe the result of the previous command to the `sort` command (which sorts the lines), and then again to `uniq` (which takes only the elements that are not repeated).[8] Effectively, we have created a pipeline to extract the names of all the orders in the database, from Afrosoricida to Tubulidentata (a remarkable order, which today contains only the aardvark).

8. The command `uniq` is typically used in conjunction with `sort`, as it will remove duplicate lines only if they are contiguous.

```
# select 2nd column without header, unique sorted elements
$ cut -d ";" -f 2 Pacifici2013_data.csv  | tail -n +2 |
    ↳ sort | uniq
Afrosoricida
Carnivora
Cetartiodactyla
...
```

This type of manipulation of character-delimited files is very fast and effective. It is an excellent idea to master the `cut` command in order to start exploring large data sets without the need to open files in specialized programs. (Note that opening a file in a text editor might modify the contents of a file without your knowledge. Find details in section 1.9.2.)

Intermezzo 1.3

(a) If we order all species names (fifth column) of `Pacifici2013_data.csv` in alphabetical order, which is the first species? Which is the last?

(b) How many families are represented in the database?

1.6.3 Substituting Characters Using `tr`

Often we want to substitute or remove a specific character in a text file (e.g., to convert a comma-separated file into a tab-separated file). Such a one-by-one substitution can be accomplished with the command `tr`. Let's look at some examples in which we use a pipe to pass a string to `tr`, which then processes the text input according to the search term and specific options.

Substitute all characters `a` with `b`:

```
$ echo "aaaabbb" | tr "a" "b"
bbbbbbb
```

Substitute every digit in the range `1` through `5` with `0`:

```
$ echo "123456789" | tr 1-5 0
000006789
```

Substitute lowercase letters with uppercase ones:

```
$ echo "ACtGGcAaTT" | tr actg ACTG
ACTGGCAATT
```

We obtain the same result by using bracketed expressions that provide a predefined set of characters. Here, we use the set of all lowercase letters `[:lower:]` and translate into uppercase letters `[:upper:]`:

```
$ echo "ACtGGcAaTT" | tr [:lower:] [:upper:]
ACTGGCAATT
```

We can also indicate ranges of characters to substitute:

```
$ echo "aabbccddee" | tr a-c 1-3
112233ddee
```

Delete all occurrences of `a`:

```
$ echo "aaaaabbbb" | tr -d a
bbbb
```

"Squeeze" all consecutive occurrences of `a`:

```
$ echo "aaaaabbbb" | tr -s a
abbbb
```

Note that the command `tr` cannot operate on a file "in place," meaning that it cannot change a file directly. However, it can operate on a copy of the contents of a file. For instance, we can use pipes in conjunction with `cat`, `head`, `cut`, or the output redirection operator to create input for `tr`:

```
# pipe output of cat to tr
$ cat inputfile.csv | tr " " "\t" > outputfile.csv
# redirect file contents to tr
$  tr " " "\t" < inputfile.csv > outputfile.csv
```

In this example we replace all spaces within the file `inputfile.csv` with tabs. Note the use of quotes to specify the space character. The tab is indicated by `\t`. The backslash defines a *metacharacter*: it signals that the following character should not be interpreted literally, but rather represents a special code referring to a character (e.g., a tab) that is difficult to represent otherwise.

Now we can apply the command `tr` and the commands we showcased earlier to create a new file containing a subset of the data contained in `Pacifici2013_data.csv`, which we are going to use in the next section.

First, we change directory to the `sandbox`:

```
$ cd ../sandbox/
```

To recap, we were working in the directory `~/CSB/unix/data`. We then moved one directory up (`..`) to get to the directory `~/CSB/unix/`, from which we moved down into the `sandbox`.

Now we want to create a version of `Pacifici2013_data.csv` containing only the `Order`, `Family`, `Genus`, `Scientific_name`, and `AdultBodyMass_g` (columns 2–6). Moreover, we want to remove the header, sort the lines according to body mass (with larger critters first), and have the values separated by spaces. This sounds like an awful lot of work, but we're going to see how this can be accomplished by piping a few commands together.

First, let's remove the header:

```
$ tail -n +2 ../data/Pacifici2013_data.csv
```

Then, take only columns 2–6:

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ";"
    ↳ -f 2-6
```

Now, substitute the current delimiter (`;`) with a space:

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ";"
    ↳ -f 2-6 | tr ";" " "
```

To sort the lines according to body size, we need to exploit a few of the options for the command `sort`. First, we want to sort numbers (option `-n`);

second, we want larger values first (option `-r`, reverse order); finally, we want to sort the data according to the sixth column (option `-k 6`):

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ";"
    ↪ -f 2-6 | tr ";" " " | sort -r -n -k 6
```

That's it. We have created our first complex pipeline. To complete the task, we redirect the output of our pipeline to a new file called `BodyM.csv`:

```
$ tail -n +2 ../data/Pacifici2013_data.csv | cut -d ";"
    ↪ -f 2-6 | tr ";" " " | sort -r -n -k 6 > BodyM.csv
```

You might object that the same operations could have been accomplished with a few clicks by opening the file in a spreadsheet editor. However, suppose you have to repeat this task many times; for example, you have to reformat every file that is produced by a laboratory device. Then it is convenient to automate this task such that it can be run with a single command. This is exactly what we are going to do in section 1.7.

Similarly, suppose you need to download a large CSV file from a server, but many of the columns are not needed. With `cut`, you can extract just the relevant columns, reducing download time and storage.

1.6.4 Wildcards

Wildcards are special symbols that work as placeholders for one or more characters. The *star wildcard* (*) stands for zero or more characters with the exception of a leading dot. Unix uses a leading dot for hidden files, so this means that hidden files are ignored in a search using this wildcard (show hidden files using `ls -a`). A question mark (?) is a placeholder for any single character, again with the exception of a leading dot.

Let's look at some examples in the directory `CSB/unix/data/miRNA`:

```
# change into the directory
$ cd ~/CSB/unix/data/miRNA
# count the numbers of lines in all the .fasta files
```

```
$ wc -l *.fasta
     714 ggo_miR.fasta
    5176 hsa_miR.fasta
     166 ppa_miR.fasta
    1320 ppy_miR.fasta
    1174 ptr_miR.fasta
      20 ssy_miR.fasta
    8570 total
# print the first two lines of each file
# whose name starts with pp
$ head -n 2 pp*
==> ppa_miR.fasta <==
>ppa-miR-15a MIMAT0002646
UAGCAGCACAUAAUGGUUUGUG

==> ppy_miR.fasta <==
>ppy-miR-569 MIMAT0016013
AGUUAAUGAAUCCUGGAAAGU

# determine the type of every file that has
# an extension with exactly three letters
$ file *.???
```

1.6.5 Selecting Lines Using grep

grep is a powerful command that finds all the lines of a file that match a given pattern. You can return or count all occurrences of the pattern in a large text file without ever opening it. grep is based on the concept of regular expressions, which we will cover in depth in chapter 5.

We explore the basic features of grep using the file we created in section 1.6.3. The file contains data on thousands of species:

```
$ cd ~/CSB/unix/sandbox
$ wc -l BodyM.csv
5426 BodyM.csv
```

Let's see how many wombats (family Vombatidae) are contained in the data. To display the lines that contain the term "Vombatidae" we execute grep with two arguments—the search term and the file that we want to search in:

```
$ grep Vombatidae BodyM.csv
Diprotodontia Vombatidae Lasiorhinus Lasiorhinus krefftii
    ↪ 31849.99
Diprotodontia Vombatidae Lasiorhinus Lasiorhinus latifrons
    ↪  26163.8
Diprotodontia Vombatidae Vombatus Vombatus ursinus 26000
```

Now we add the option `-c` to count the lines that contain a match:

```
$ grep -c Vombatidae BodyM.csv
3
```

Next, we have a look at the genus *Bos* in the data file:

```
$ grep Bos BodyM.csv
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
Cetartiodactyla Bovidae Bos Bos mutus 650000
Cetartiodactyla Bovidae Bos Bos javanicus 635974.3
Cetartiodactyla Bovidae Boselaphus Boselaphus tragocamelus
    ↪  182253
```

Besides all the members of the *Bos* genus, we also match one member of the genus *Boselaphus*. To exclude it, we can use the option `-w`, which prompts `grep` to match only full words:

```
$ grep -w Bos BodyM.csv
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
Cetartiodactyla Bovidae Bos Bos mutus 650000
Cetartiodactyla Bovidae Bos Bos javanicus 635974.3
```

Using the option `-i` we can make the search case insensitive (it will match both upper- and lowercase instances):

```
$ grep -i Bos BodyM.csv
Proboscidea Elephantidae Loxodonta Loxodonta africana
    ↪ 3824540
```

```
Proboscidea Elephantidae Elephas Elephas maximus 3269794
Cetartiodactyla Bovidae Bos Bos sauveli 791321.8
Cetartiodactyla Bovidae Bos Bos gaurus 721000
...
```

Sometimes, we want to know which lines precede or follow the one we want to match. For example, suppose we want to know which mammals have body weight most similar to the gorilla (*Gorilla gorilla*). The species are already ordered by size (see section 1.6.3), thus we can simply print the two lines before the match using the option `-B 2` and the two lines after the match using `-A 2`:

```
$ grep -B 2 -A 2 "Gorilla gorilla" BodyM.csv
Cetartiodactyla Bovidae Ovis Ovis ammon 113998.7
Cetartiodactyla Delphinidae Lissodelphis Lissodelphis
    ↳ borealis 113000
Primates Hominidae Gorilla Gorilla gorilla 112589
Cetartiodactyla Cervidae Blastocerus Blastocerus
    ↳ dichotomus 112518.5
Cetartiodactyla Iniidae Lipotes Lipotes vexillifer
    ↳ 112138.3
```

Use option `-n` to show the line number of the match. For example, the gorilla is the 164th largest mammal in the database:

```
$ grep -n "Gorilla gorilla" BodyM.csv
164:Primates Hominidae Gorilla Gorilla gorilla 112589
```

To print all the lines that do not match a given pattern, use the option `-v`. For instance, we want to find species of the genus *Gorilla* other than *Gorilla gorilla*. We can pipe the result of matching all members of the genus *Gorilla* to a second `grep` statement that excludes the species *Gorilla gorilla*:

```
$ grep Gorilla BodyM.csv | grep -v gorilla
Primates Hominidae Gorilla Gorilla beringei 149325.2
```

To match one of several strings, use `grep "[STRING1]\|[STRING2]"`:

```
$ grep -w "Gorilla\|Pan" BodyM.csv
Primates Hominidae Gorilla Gorilla beringei 149325.2
Primates Hominidae Gorilla Gorilla gorilla 112589
Primates Hominidae Pan Pan troglodytes 45000
Primates Hominidae Pan Pan paniscus 35119.95
```

You can use `grep` on multiple files at a time! Simply list all the files that you want to search (or use wildcards to specify multiple file names). Finally, use the recursive search option `-r` to search for patterns within all the files in a directory. For example,

```
$ cd ~/CSB/unix
# search recursively in the data directory
$ grep -r "Gorilla" data
```

1.6.6 Finding Files with find

The `find` command is the command-line program to locate files in your system. You can search by file name, owner, group, type, and other criteria. For example, find the files and subdirectories that are contained in the `unix/data` directory:

```
# current directory is the unix sandbox
$ find ../data
../data
../data/Buzzard2015_data.csv
../data/Pacifici2013_data.csv
../data/Gesquiere2011_about.txt
../data/Gesquiere2011_data.csv
../data/Saavedra2013
...
```

To count occurrences, we can pipe to `wc -l`:

```
$ find ../data | wc -l
77
```

Now we can use find to match particular files. First, we specify where to search: this could be either an absolute path (e.g., /home/YOURNAME/CSB/unix/data) or a relative one (e.g., ../data, provided we're in unix/sandbox).

If we want to match a specific file name, we can use the option -name:

```
$ find ../data -name "n30.txt"
../data/Saavedra2013/n30.txt
```

To exploit the full power of find, we use wildcards.[9] For example, use the * wildcard to find all the files whose names contain the word about; the option -iname ignores the case of the file name:

```
$ find ../data -iname "*about*"
../data/Gesquiere2011_about.txt
../data/Buzzard2015_about.txt
...
```

You can specify the depth of the search by limiting it to, for example, only the directories immediately descending from the current one. See the difference between

```
$ find ../data -name "*.txt" | wc -l
64 # depending on your system
```

and

```
$ find ../data -maxdepth 1 -name "*.txt" | wc -l
5
```

which excluded all files in subdirectories. You can exclude certain files:

```
$ find ../data -not -name "*about*" | wc -l
72
```

9. See section 1.6.4 for an introduction to wildcards.

or find only directories:

```
$ find ../data -type d
../data
../data/miRNA
../data/Saavedra2013
```

Intermezzo 1.4

(a) Navigate to `CSB/unix/sandbox`. Without navigating to a different location, find a CSV file that contains `Dalziel` in its file name and is located within the `CSB` directory. Copy this file to the Unix `sandbox`.
(b) Print the first few lines on the screen to check the structure of the data. List all unique cities in column `loc` (omit the header). How often does each city occur in the data set?
(c) The fourth column reports cases of measles. What is the maximum number of cases reported for Washington, DC?
(d) What is the maximum number of reported measles cases in the entire data set? Where did this occur?

1.6.7 Permissions

In Unix, each file and directory has specific security attributes specifying who can read (`r`), write (`w`), execute (`x`), or do nothing (`-`) with the file or directory. These permissions are specified for three entities that may wish to manipulate the file (owner, specific group, and others). The group level is useful for assigning permissions to a specific group of users (e.g., administrators, developers) but not everyone else.

Typing `ls -l` lists the permissions of each file or subdirectory at the beginning of the line. Each permission is represented by a 10-character notation. The first character refers to the file type and is not related to permissions (`-` means file, `d` stands for directory). The last 9 are arranged in groups of 3 (triads) representing the ownership groups (owner, group, others). For example, when a file has the permission `-rwxr-xr--`, the owner of this file can read, write, and execute the file (`rwx`), the group can read and execute (`r-x`), while everyone else can only read (`r--`).

The commands `chmod` and `chown` change the permissions and ownership of a file, respectively:

```
# create a file in the unix sandbox
$ touch permissions.txt
# look at the current permissions
# (output will be different on your machine)
$ ls -l
-rw-r--r--  1 mwilmes  staff  0 Aug 15 09:47 permissions.
    ↳ txt
# change permissions (no spaces between mode settings)
$ chmod u=rwx,g=rx,o=r permissions.txt
# look at changes in permissions
$ ls -l
-rwxr-xr--  1 mwilmes  staff  0 Aug 15 09:48 permissions.
    ↳ txt
# take execute permission away from group,
# and add write rights to others
$ chmod g-x,o+w permissions.txt
$ ls -l
-rwxr--rw-  1 mwilmes  staff  0 Aug 15 09:49 permissions.
    ↳ txt
```

Some operations, such as changing the ownership of a file or directory, or installing new software, can be performed only by the administrator of the machine. You, however, can do so by typing the word `sudo` (substitute user do) in front of the command. The system will request a password and, if you are authorized to use the `sudo` command, you grant yourself administrator rights.

When you download and install new software, the system will often request the administrator's password. Pay attention to the trustworthiness of the source of the software before confirming the installation, as you may otherwise install malicious software.

Here is an example of changing the file permissions for a directory recursively (i.e., for all subdirectories and files):

```
# create a directory with a subdirectory
$ mkdir -p test_dir/test_subdir
# look at permissions
$ ls -l
drwxr-xr-x  3 mwilmes  staff  102 Aug 15 10:59 test_dir
# change owner of directory recursively using -R
```

```
$ sudo chown -R sallesina test_dir/
# check for new ownership
$ ls -l
drwxr-xr-x  3 sallesina  staff  102 Aug 15 11:01 test_dir
```

1.7 Basic Scripting

Once a pipeline is in place, it is easy to turn it into a *script*. A script is a text file containing a list of several commands. The commands are then executed one after the other, going through the pipeline in an automated manner. To illustrate the concept, we are going to turn the pipeline in section 1.6.3 into a script.

First, we need to create a file for our Unix script that we can edit using a text editor. The typical extension for a file with shell commands is `.sh`. In this example, we want to create the file `ExtractBodyM.sh`, which we can open using our favorite text editor. Create an empty file, either in the editor, or using `touch`:

```
$ touch ExtractBodyM.sh
```

Open the file in a text editor. In Ubuntu you can use, for example, gedit:

```
$ gedit ExtractBodyM.sh &
```

In OS X, calling `open` will open the file with the default text editor:[10]

```
$ open ExtractBodyM.sh &
```

The "ampersand" (`&`) at the end of the line prompts the terminal to open the editor in the background, so that you can still use the same shell while working on the file. Windows users can use any text editor.[11]

Now copy the pipeline that we built throughout the previous sections into the file `ExtractBodyM.sh`. For now, make sure that it is one long line:

10. Use option `-a` to choose a specific editor (e.g., `open -a emacs ExtractBodyM.sh &`).

11. Make sure, however, that the editor can save files with the Unix line terminator (LF), otherwise the scripts will not work correctly (details in section 1.9.2). Here's a list of suitable editors: computingskillsforbiologists.com/texteditors.

```
tail -n +2 ../data/Pacifici2013_data.csv | cut -d ";"
    -f 2-6 | tr ";" " " | sort -r -n -k 6 > BodyM.csv
```

and save the file. To run the script, call the command `bash` and the file name:

```
$ bash ExtractBodyM.sh
```

It is a great idea to immediately write comments for the script, to help you remember what the code does. You can add comments using the hash symbol (#):

```
# take a CSV file delimited by ";"
# remove the header
# make space separated
# sort according to the 6th (numeric) column
# in descending order
# redirect to a file
tail -n +2 ../data/Pacifici2013_data.csv | cut -d ";"
    -f 2-6 | tr ";" " " | sort -r -n -k 6 > BodyM.csv
```

As it stands, this script is very specific: both the input file and the output file names are fixed (*hard coded*). It would be better to leave these names to be decided by the user so that the script can be called for any file with the same format. This is easy to accomplish within the Bash shell: simply use generic arguments (i.e., variables), indicated by the dollar sign ($), followed by the variable name (without a space). Here, we use the number of the argument as the variable name. When the script is run, the generic arguments within the script are replaced by the specific argument that the user supplied when executing the script.

Let's change our script `ExtractBodyM.sh` accordingly:

```
# take a CSV file delimited by ";" (first argument)
# remove the header
# make space separated
```

```
# sort according to the 6th (numeric) column
# in descending order
# redirect to a file (second argument)
tail -n +2 $1 | cut -d ";" -f 2-6 | tr ";" " " | sort -r -n
    -k 6 > $2
```

The file name (i.e., `../data/Pacifici2013_data.csv`) and the result file (i.e., `BodyM.csv`) have been replaced by `$1` and `$2`, respectively. Now you can launch the modified script from the command line by specifying the input and output files as arguments:

```
$ bash ExtractBodyM.sh ../data/Pacifici2013_data.csv BodyM
    ↳ .csv
```

The final step is to make the script directly executable so that you can skip invoking Bash. We can do so by changing the permissions of the file,

```
$ chmod +rx ExtractBodyM.sh
```

and adding a special line at the beginning of the script telling Unix where to find the program (in this case `bash`[12]) to execute the script:

```
#!/bin/bash

# the previous line is not a comment, but a special line
# telling where to find the program to execute the script;
# it should be your first line in all Bash scripts

# function of script:
# take a CSV file delimited by ";" (first argument)
# remove the header
# make space separated
```

12. If you don't know where the program `bash` is, you can find out by running `whereis bash` in your terminal.

```
# sort according to the 6th (numeric) column
# in descending order
# redirect to a file (second argument)
tail -n +2 $1 | cut -d ";" -f 2-6 | tr ";" " " | sort -r -n
    -k 6 > $2
```

Now, this script can be invoked as

```
$ ./ExtractBodyM.sh ../data/Pacifici2013_data.csv BodyM.
    ↳ csv
```

Note the ./ in front of the script's name in order to execute the file.

The long Unix pipe that we constructed over the last few pages can be complicated to read and understand. It is therefore convenient to break it into smaller pieces and save the individual output of each part as a temporary file that can be deleted as a last step in the script:

```
#!/bin/bash
# function of script:
# take a CSV file delimited by ";" (first argument)
# remove the header
# make space separated
# sort according to the 6th (numeric) column
# in descending order
# redirect to a file (second argument)

# remove the header
tail -n +2 $1 > $1.tmp1
# extract columns
cut -d ";" -f 2-6 $1.tmp1  > $1.tmp2
# make space separated
tr ";" " " < $1.tmp2 > $1.tmp3
# sort and redirect to output
sort -r -n -k 6 $1.tmp3 > $2
# remove temporary, intermediate files
rm $1.tmp*
```

This is much more readable, although a little more wasteful, as it creates temporary files only then to delete them. Using intermediate, temporary files,

however, allows scripts to be "debugged" easily—just comment the last line out and inspect the temporary files one by one to investigate at which point you obtained an unwanted result.

1.8 Simple `for` Loops

A `for` loop allows us to repeat a task with slight variations. For instance, a loop is very useful when you need to perform an identical task on multiple files, or when you want to provide different input arguments for the same command. Instead of writing code for every instance separately, we can use a loop.

As a first example, we want to display the first two lines of all `.fasta` files in the directory `CSB/unix/data/miRNA`. We first change the directory and execute the `ls` command to list its contents:

```
$ cd ~/CSB/unix/data/miRNA
$ ls
ggo_miR.fasta   hsa_miR.fasta   ppa_miR.fasta   ...
```

The directory contains six `.fasta` files with miRNA sequences of different *Hominidae* species. Now we want to get a quick overview of the contents of the files. Instead of individually calling the `head` command on each file, we can access multiple files by writing a `for` loop:

```
$ for file in ggo_miR.fasta hsa_miR.fasta
        do head -n 2 $file
   done
>ggo-miR-31 MIMAT0002381
GGCAAGAUGCUGGCAUAGCUG
>hsa-miR-576-3p MIMAT0004796
AAGAUGUGGAAAAAUUGGAAUC
```

Here we created a variable (`file`) that stands in for the actual file names that are listed after the `in`. Instead of listing all files individually after the `in`, we can also use wildcards to consider all `.fasta` files in the directory:

```
$ for file in *.fasta
        do head -n 2 $file
   done
```

```
>ggo-miR-31 MIMAT0002381
GGCAAGAUGCUGGCAUAGCUG
>hsa-miR-576-3p MIMAT0004796
AAGAUGUGGAAAAAUUGGAAUC
...
```

The actual statement (i.e., what to do with the variable) is preceded by a `do`. As shown in section 1.7, the variable is invoked with a $ (dollar sign). The statement ends with `done`. Instead of this clear coding style that spans multiple lines, you may also encounter loops written in one line, using a ; as command terminator instead of line breaks.

In our second example, we call a command with different input variables. Currently, the files in `CSB/unix/data/miRNA` provide files that contain different miRNA sequences per species. However, we might need files that contain all sequences of different species per type of miRNA. We can accomplish this by using the command `grep` in a `for` loop instead of writing code for every type of miRNA separately:

```
$ for miR in miR-208a miR-564 miR-3170
        do grep $miR -A1 *.fasta > $miR.fasta
   done
```

We have created the variable `miR` that cycles through every item in the list that is given after the `in` (i.e., types of miRNA). In every iteration of the loop, one instance of the variable is handed to `grep`. We used the same variable again to create appropriate file names.

Let's have a look at the head of one of the files that we have created:

```
$ head -n 5 miR-564.fasta
hsa_miR.fasta:>hsa-miR-564 MIMAT0003228
hsa_miR.fasta-AGGCACGGUGUCAGCAGGC
--
ppy_miR.fasta:>ppy-miR-564 MIMAT0016009
ppy_miR.fasta-AGGCACGGUGGCAGCAGGC
```

We can see that the output of `grep` is the name of the original file where a match was found, followed by the line that contained the match. The `-A1` option of `grep` also returned the line after the match (i.e., the sequence).

Knowing how to perform such simple loops using the Bash shell is very beneficial. However, Bash has a rather idiosyncratic syntax that does not lend itself well to performing more complex programming tasks. We will therefore cover general programming comprehensively in chapter 3, which introduces a programming language with a friendlier syntax, Python.

1.9 Tips, Tricks, and Going beyond the Basics

1.9.1 Setting a `PATH` in `.bash_profile`

Have you come across the error message `command not found`? You may have simply mistyped a command, or tried to invoke a program that is not installed on your machine. Maybe, however, your computer doesn't know the location of a program, in which case this can be resolved by adding the path (location) of a program to the `PATH` variable. Your computer uses `$PATH` to search for corresponding executable files when you invoke a command in the terminal. To inspect your path variable, type

```
# print path variable to screen
$ echo $PATH
```

You can append a directory name (i.e., location of a program) to your `PATH` by editing your `.bash_profile`. This file customizes your Bash shell (e.g., sets the colors of your terminal or changes the command-line prompt). If this hidden file does not exist in your home directory (check with `ls -a`), you can simply create it. Here is how to append to your computer's `PATH` variable (`$PATH`):

```
# add path to a program to computer's PATH variable
$ export PATH=$PATH:[PATH TO PROGRAM]
```

You can use `which` to identify the path to a program:

```
# identify the path to the grep command
$ which grep
/usr/bin/grep
```

Note that the order of elements in the `PATH` matters. If you have several versions of a program installed on your machine, the one that is found first (i.e., its location is represented earlier in the `PATH`) will be invoked.

1.9.2 Line Terminators

In text files, line terminators are represented by *nonprinting characters*. These are special characters that indicate white space or special formatting (e.g., space, tab, line break, nonbreaking hyphen). Unless you explicitly ask your editor to display them, they will not print to the screen (hence nonprinting). Unfortunately, different platforms use different symbols to indicate line breaks. While Unix-like systems use a line feed (`\n`), Windows uses a carriage return and linefeed combination (`\r\n`).

Many text editors autodetect the line terminator and display your file correctly (i.e., alter your file). However, if you encounter text that looks like a single long line, or displays `^M` where a line break should be, you might want to change the nonprinting symbol for the line terminator.

Also, be careful when copying text between files with different line terminators—you might end up with a hybrid that will be difficult to deal with later on. When working with an unknown text file, use `file` to determine the type of line terminator before opening the file.

1.9.3 Miscellaneous Commands

Without much ado we want to provide some pointers to interesting topics and commands that might come in handy. Please refer to the documentation for usage and examples. Note that some commands are not available on all platforms, but can be installed.

`history`	List the last commands you executed.[13]
`time [COMMAND]`	Time the execution of a command.
`wget [URL]`	Download the web page at `[URL]`.[14]
`open`	Open file or directory with default program; use `xdg-open` in Ubuntu or `start` in Windows Git Bash.
`rsync`	Synchronize files locally or remotely.
`tar` and `zip`	(Un)compress and package files and directories.
`awk` and `sed`	Powerful command-line text editors for much more complex text manipulation than `tr`.

13. In Git Bash all commands are listed.

14. Available in Ubuntu; for OS X look at `curl`, or install `wget` (see computingskillsforbiologists.com/wget).

`xargs`	Pass a list of arguments to other commands; for example, create a file for each line in `files.txt`: `cat files.txt | xargs touch`

1.10 Exercises

1.10.1 Next Generation Sequencing Data

In this exercise we work with next generation sequencing (NGS) data. Unix is excellent at manipulating the huge FASTA files that are generated in NGS experiments.

FASTA files contain sequence data in text format. Each sequence segment is preceded by a single-line description. The first character of the description line is a "greater than" sign (>).[15]

The NGS data set we will be working with was published by Marra and DeWoody (2014), who investigated the immunogenetic repertoire of rodents. You will find the sequence file `Marra2014_data.fasta` in the directory `CSB/unix/data`. The file contains sequence segments (contigs) of variable size. The description of each contig provides its length, the number of reads that contributed to the contig, its isogroup (representing the collection of alternative splice products of a possible gene), and the isotig status.

1. Change directory to `CSB/unix/sandbox`.
2. What is the size of the file `Marra2014_data.fasta`?[16]
3. Create a copy of `Marra2014_data.fasta` in the `sandbox` and name it `my_file.fasta`.
4. How many contigs are classified as `isogroup00036`?
5. Replace the original "two-spaces" delimiter with a comma.
6. How many unique isogroups are in the file?
7. Which contig has the highest number of reads (`numreads`)? How many reads does it have?

1.10.2 Hormone Levels in Baboons

Gesquiere et al. (2011) studied hormone levels in the blood of baboons. Every individual was sampled several times.

15. See computingskillsforbiologists.com/fasta for more details on the FASTA file format.

16. Note that the original sequence file is much larger! We truncated the file to 1% of its original size to facilitate the download.

1. How many times were the levels of individuals 3 and 27 recorded?
2. Write a script taking as input the file name and the ID of the individual, and returning the number of records for that ID.
3. [**Advanced**][17] Write a script that returns the number of times each individual was sampled.

1.10.3 Plant–Pollinator Networks

Saavedra and Stouffer (2013) studied several plant–pollinator networks. These can be represented as rectangular matrices where the rows are pollinators, the columns plants, a 0 indicates the absence and 1 the presence of an interaction between the plant and the pollinator.

The data of Saavedra and Stouffer (2013) can be found in the directory `CSB/unix/data/Saavedra2013`.

1. Write a script that takes one of these files and determines the number of rows (pollinators) and columns (plants). Note that columns are separated by spaces and that there is a space at the end of each line. Your script should return

```
$ bash netsize.sh ../data/Saavedra2013/n1.txt
Filename: ../data/Saavedra2013/n1.txt
Number of rows: 97
Number of columns: 80
```

2. [**Advanced**][18] Write a script that prints the numbers of rows and columns for each network:

```
$ bash netsize_all.sh
../data/Saavedra2013/n10.txt 14 20
../data/Saavedra2013/n11.txt 270 91
../data/Saavedra2013/n12.txt 7 72
../data/Saavedra2013/n13.txt 61 17
...
```

17. This task requires being able to capture the output of a command within a script (see, e.g., computingskillsforbiologists.com/captureoutput) and writing a "loop" iterating through all IDs (see, e.g., computingskillsforbiologists.com/bashloops).

18. This exercise requires writing a loop within a script.

3. Which file has the largest number of rows? Which has the largest number of columns?

1.10.4 Data Explorer

Buzzard et al. (2016) collected data on the growth of a forest in Costa Rica. In the file `Buzzard2015_data.csv` you will find a subset of their data, including taxonomic information, abundance, and biomass of trees.

1. Write a script that, for a given CSV file and column number, prints
   - the corresponding column name;
   - the number of distinct values in the column;
   - the minimum value;
   - the maximum value.

   For example, running the script with

```
$ bash explore.sh ../data/Buzzard2015_data.csv 7
```

should return

```
Column name:
biomass
Number of distinct values:
285
Minimum value:
1.048466198
Maximum value:
14897.29471
```

1.11 References and Reading

Books

J. Peek et al., *Unix Power Tools*, O'Reilly Media, 2005.
Tips & tricks to help you think creatively about Unix and solve your own problems.

A. Robbins & N. H. F. Beebe, *Shell Scripting*, O'Reilly Media, 2002.
Learn how to combine the fundamental Unix text and file processing commands to crunch data and automate repetitive tasks.

Online Resources

There are many good tutorials on Unix and shell scripting. Azalee Bos of Software Carpentry has a nice video tutorial on YouTube (shell starts at minute 11):
computingskillsforbiologists.com/unixvideo.

Also check out the material provided by Software Carpentry:
computingskillsforbiologists.com/unixtutorials.

The website SHELLdorado contains a list of useful shell scripts, as well as a series of tips & tricks:
shelldorado.com.

There is a series of tutorials on basic Unix commands:
computingskillsforbiologists.com/shelltutorials.

Many Unix commands are listed on Wikipedia:
computingskillsforbiologists.com/unixcommands.