Appendix A

Using the Command Line

In this appendix, we introduce the command line interface and give brief tutorials on the various functions it can perform. We will also include an overview of a few more advanced techniques to give you an idea of the range of uses of the shell. Though it may seem like just a means to an end, effective use of the command line is one of the most important skills you can develop when learning data analysis. Don't worry about memorizing all the commands and functions given here; this appendix is meant to serve as a reference for you to use throughout the text. As you use the command line more, you will figure out what commands are the most useful for you and worth remembering.

A.1 What is the command line?

For most functions we use computers for, we interact with the operating system through what is known as a **Graphical User Interface** (GUI, pronounced "gooey"). This is the environment in which the mouse, icons, toolbars, etc. live and can be manipulated in. Most operating systems do their best to make this user interface as simple as possible, so that it's easy to perform a wide range of operations with the minimal number of actions. For some tasks though, it is faster and more efficient to work more directly with the operating system through a **Command Line Interface** (CLI). A CLI is a text-based interface where the user gives line-by-line commands to the system through a command prompt. The form and syntax of allowed commands is determined by the **shell**. The shell is the medium through which the user communicates with the operating system; it translates the user's commands into instructions the operating system can understand and execute. There are several different shells that can be used for a given operating system, and they will differ in command notation and syntax. To check which shell is being run in a command line session, simply enter the command `echo $SHELL` into the command prompt, and it will return the shell being sourced for that session (either sh, csh, ksh, or bash). Here, we will focus on one of the most common Unix/Linux (and Windows 10) shells, *bash*.

The name *bash* originates from the abbreviation of "Bourne again shell," which is a play on words referring to Stephen Bourne, the inventor of the first popular UNIX shell which bash eventually replaced. Bash has been widely used since its initial release in 1989, and is still included as the default shell for most Unix (Mac) and Linux computers. It was even made available for Windows 10 users in 2016. Bash offers users many useful features, including command line editing modes, job control, and extensive programmability. All in all, bash is one of the most well-rounded and versatile shells available. Though it will take some practice, becoming fluent in bash can drastically increase your productivity and make your life much easier.

A.2 Basic bash commands

Now that we understand the role the shell plays in how we give commands to the operating system, let's examine the anatomy of these commands. Shell commands can come in a variety of forms, but they all follow the general structure of a single **command** often followed by a list of **parameters**. These parameters are often file names, directory paths, or command options. Depending on the command being used, there may be several parameters needed or none at all. Code Example A.1 gives the form and output of a few bash commands, including the one you will use the most, `ls` and `cd`. Before examining these commands further, let's first establish what exactly we mean by files and directories.

File is a general term referring to some data-containing object. For this discussion, however, assume files come in three types: source files, executable files, and directories. Source files (usually referred to as text files) contain lines of text that are readable by the user. Normally, these are the type of files you will write code in. Executable files, on the other hand, are special files that can be called as commands. These files may be legible by the user, but are usually in a language only readable by the operating system. Both these files types are organized and stored into directories, which act as folders that can contain files of any type. Directories can even contain other directories, which naturally leads to a file systems having a hierarchical structure with a nest of directories and subdirectories.

To unambiguously refer to a file's location on a computer, you not only need to designate what subdirectory the file is in, but also what directory that subdirectory is in, and so on until you are at the top directory. This list of directories is called a **path**. For some purposes, however, you may only need to refer to files contained within a particular directory (or its subdirectories), so you would be repeating information every time you had to include the path to that directory. To remove this potential redundancy, the shell has the concept of a **working directory**. This can be thought of as *where you are* in the file system, and all paths without a leading slash (or tilde) are interpreted relative to this directory. You can always check your current directory with the `pwd` command, which prints the full path to the current directory. To change your working directory, you can use the `cd` command. This command takes the path of a directory as an argument, and then makes it the current directory. This path can refer to a subdirectory of the directory you are currently in, or it can be the full path to any directory. If you would like to move to a directory that is several directories below you, you can chain together names in a `cd` command path rather than having to execute the command multiple times. For example, `cd dir1/dir2` is the same as `cd dir1` followed by `cd dir2`. The `cd` command doesn't From any directory, you can always enter `cd` without any arguments, and you will be returned to your home directory.

Once you have identified what directory you're in, you may be interested in the contents of this directory. To determine this, you can use the `ls` command, which returns a list of all the files in a directory. Generally, you give the path to the directory you would like the contents of, but you can also give no argument, and it will list the contents of the current directory. Sometimes when searching for a particular file in a directory with a large number of contents, it can difficult to find one file name in the long list given. If you know any part of the file name, however, you can reduce the list size using **wildcards**. These are special characters that represent multiple characters. The most useful of these is "*", which bash interprets as any string of characters. For example, the command `ls a*` will return a list of all the file names that begin with the letter a in the current directory. Alternatively, `ls some_path/*.pdf` will return a list of all files that end in ".pdf" in the directory `some_path`.

Returning to Code Example A.1, we can now better understand the outputs given by the commands. We start in our home directory and `cd` into Desktop and check the path of this directory. We then list the files in this directory. From there, we check the contents of the Lab directory with `ls Lab`. We then `cd` into the Lab directory, and again list its contents (which returns the same list as when we did it from the Desktop). We then `cd` out of Lab back into Desktop. Notice that this is done using the ".." path. This path represents the directory *above* your current directory; this simply means the directory that the current directory is in. So in the case of the Lab directory (which has the path /Users/AlexDorsett/Desktop/Lab), entering `cd`

Code Example A.1: Outputs of a few bash commands that help with managing directories.

```
[03:23:00]~/$ cd Desktop
[03:23:09]~/Desktop$ pwd
/Users/AlexDorsett/Desktop
[03:23:11]~/Desktop$ ls
Alex    Books   Lab     Nuclear    Project125    UCSB
[03:23:13]~/Desktop$ ls Lab/
HiggsSig ReReco17 TrigPlots update.py
MiniIso    Slides     data17     trigs
[03:23:15]~/Desktop$ cd Lab/
[03:23:16]~/Desktop/Lab$ ls
HiggsSig ReReco17 TrigPlots update.py
MiniIso    Slides     data17     trigs
[03:23:18]~/Desktop/Lab$ cd ..
[03:23:20]~/Desktop$
```

`..` returns us to the Desktop. You may notice in Code Example A.1 that the command prompt changes depending on the current directory (and time). We will get to how to customize this later, but it is worth noting that when we are in our home directory, the path printed is simply `~/`. This is because the ~(tilde) character is used to abbreviate the path to the home directory (`/Users/AlexDorsett` in this case). The home directory comes up a lot in paths, so using the ~shorthand is an easy way to avoid always having to remember (and type out) its path each time you need it.

Using the command line can seem like a daunting task at times. There are many different commands and options, and if you don't enter everything in just right, the shell will (usually) not complete the intended task. Fortunately, bash includes several features intended to reduce user mistakes, as well as to make it easier to fix them when they do occur. One of these features that will make completing commands much faster is **tab completion**. When specifying the name of a file (either in a path or as the input for a command) you don't need to always type out the full name, but just enough for it to be specified unambiguously. If you have done this, you can then use the tab key to complete the name for you. For example, say you are in a directory with 5 subdirectories (with names `Chestnut`, `Hazel`, `Ivy`, `Cherry`, and `Orange`) and you would like to list the contents of two of them, say `Hazel` and `Cherry`. To list `Hazel`, all you would need to type is `ls H` and then press tab and bash will automatically complete the name (Hazel is the only file that begins with H), and also add a forward slash after in case you were going to continue the path. In the case of `Cherry`, however, if you type `ls C` and then press tab, bash will only complete the name up to the first 3 letters (the command will now read `ls Che`). This is because there are two files in this path that begin with 'Che', and when C is specified, bash 'knows' this can only be referring to one of these two files (no other files in this path begin with C). Rather than do nothing when tab is pressed (since you haven't clearly identified a file), bash will complete the name up to the point where they differ. If you press tab once more, bash will list all the files that begin with the specified string (in this case `Cherry` and `Chestnut`). You could then type 'r' and be able to tab-complete the name completely to `ls Cherry/`.

Though tab completion can help to make sure that you never mistype a file name, command line mistakes will inevitably occur. Alternatively, you may want to rerun a particularly long command with a different option or input. For these cases, bash allows you to quickly access past commands with **command history editing**. When at the command line, simply use the up/down arrow keys to cycle through previous commands you have entered, then use the left/right arrow keys to move through and edit these commands. Additionally, there are a few extra commands that are useful when navigating through previous commands. For example, if the mistake or change is at the beginning of a long command, typing CTRL-a will move the cursor to the beginning of the current line (CTRL-e goes to the end). If you would like to keep the first half of a command, CTRL-k deletes all text from the cursor to the end of the line. Between tab completion and command line editing, bash greatly reduces the amount of unnecessary work the user has to do when

Table A.1: Common bash commands and options. For commands that have them, multiple options can be combined into one argument (`-al` is the same as `-a -l`).

Purpose	Command	op	Description
Directory Management	`ls [ops] [path]`		List the files in [path]
		`-a`	Include hidden files
		`-l`	List files in long format
	`pwd`		Display the current directory path
	`cd [path]`		Change to another directory ([path])
	`mkdir [ops] [path]`		Create the directory [path]
		`-p`	Create intermediate directories as required
	`rmdir [path]`		Delete the directory [path] (must be empty)
File Management	`mv [ops] [path1] [path2]`		Move a file from [path1] to [path2]
		`-n`	Do not overwrite file at [path2] if it already exists
	`cp [path1] [path2]`		Copy a file from [path1] to [path2]
		`-R`	Copy entire directory with its contents to [path2]
	`rm [ops] [path]`		Delete the file at [path]
		`-r`	Remove directories as well as files
File Reading & Searching	`cat [ops] [file]`		Print contents of a file to the command window
		`-n`	Number the output lines
	`grep [ops] "text" [path]`		Print all instances of "text" for files in [path]
		`-C i`	Display i lines before/after each instance of "text"
	`diff [file1] [file2]`		List all the differences between two files
		`-y`	Output both files side-to-side

using the command line.

A.3 Editing files

When it comes to actually writing and editing code, you will need a way to open and read files. Most file formats you are used to dealing with (.pdf, .mp3, .gif, etc.) can only be read or interpreted by programs designed to read these file types. If you were to open these files in a text editor, you would just see a lot of indecipherable text. When writing code, however, the source files will simply contain a list of commands in whatever language the program is written. Reading these programs with a text editor is straightforward, and making changes is easy. Though this editing can be accomplished by a basic text editor, there are a wide range of programs that offer extra benefits and utilities with coding in mind. Some of these programs can be run directly from the command line (such as vim or emacs), and others come with a completely separate interface (such as Xcode or Sublime). There are pros and cons to each of these options, but here I will focus on an editor that can be used in almost all situations, vim.

Sometimes, when remotely logging into another machine, you don't have the ability to open files locally, but rather have to edit them on the remote machine itself. This means that you are unable to open the file in a new window or external program on your computer. In cases like this, it is very useful to be able to edit the code directly from the command line. Vim offers the ability to do this, as it is already installed on most machines that you will login to. By simply typing `vim file-name`, you can view the source text of any file. Since this program runs through the command line, the mouse cannot be used to interact with the text, but you can navigate using the arrow keys (or h,j,k, and l if you are so inclined). Code Example A.2 gives a summary of the basic set of editing commands in vim. Additionally, vim provides a brief (but thorough) tutorial for users, just enter `vimtutor` into the command line of any machine with vim installed. This is a good exercise to familiarize yourself with the Vim editing environment (it can feel a bit awkward at first). If you spend the time to work through all the examples in this tutorial, you will be well on your

Code Example A.2: Basic commands in the vim text editor

```
//From Normal mode (Mode when you first open vim)
i  | Enter insert mode, where you can add/delete text
esc| Return to Normal mode
v  | Enter visual mode, where you can highlight sections of text for various functions
   y - "Yank" selected text to the clipboard (copy)
   d - remove selected text, add it to the clipboard (cut)
p  | Paste text in clipboard in front of the cursor
dd | Delete current line of text
u  | Undo the previous action (can be repeated)
:w | Save file
:q | Quit vim (:wq saves and quits)
```

way to Vim mastery.

A.4 Remote Access

Sometimes, you may be unable to complete a task on your computer. Whether it be because you require files that are too large to store on your hard drive or you need more computing power than your machine can provide, you may need to use another computer. Whatever the reason, Linux/Unix systems allow you to connect to other systems through the command line with **SS H login** (SSH stands for 'Secure Shell'). This protocol gives you the ability to remotely interface with another computer through the command line. To do this, enter `ssh [username]@[hostname]` into the command line, where `[hostname]` is either the IP address or domain name of the remote machine, and `[username]` is your registered account name on the remote system. You may then be asked for a password, depending on the security settings of the server. Once you have successfully logged in, your command line will become a shell session on the remote machine, starting in the home directory of your account. Your environment is completely determined by the remote machine, and any commands you enter will be executed there. To terminate the remote shell session, enter `exit` at any time, and you will be returned to the environment you gave the `ssh` command in.

By default, the SSH protocol only allows command line information (text inputs and outputs) to be transmitted between the user and host machine. For other types of information to be transferred–such as graphics or additional windows–permission needs to be given at the time the connection is established. This can be done by including the `-X` option when entering the `ssh` command. This will allow any windows or graphics produced during the remote shell session to be displayed on your computer. This makes it much easier to examine data and make plots on a remote machine, as you are able to examine the outputs and make adjustments without having to exit the session.

You may find yourself logging in to a remote machine quite frequently. Assuming the account you're logging into is password protected (as they usually are), you'll have to enter your password every time you login. After a while, this can get tiresome. Fortunately, the SSH protocol allows you to configure your remote account so that it will 'trust' your computer, meaning that it won't require a password when you are logging in from that computer. To accomplish this, you will need to generate a key that will be stored on both computers in specific locations. On the remote machine, create the directory `.ssh` in the home directory. On your computer, enter the commands shown in Code Example A.3. Once you have done this, you should be able to login to the remote machine without having to enter your password. It may take a bit of work to set up, but this can be a huge time-saver, and also potentially save you from having to remember the password on several different accounts.

Another important command when working with remote machines is `scp` (secure copy). This allows

Code Example A.3: Commands to enter on your computer (starting from your home directory) to generate and then transfer an ssh key to enable logging in without a password. Before entering these commands, make sure the directory ~/.ssh exists on the remote server.

```
[09:20:01]~$ mkdir .ssh
[09:20:06]~$ cd .ssh
[09:20:09]~/.ssh$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (~/.ssh/id_rsa):
...
[09:20:17]~/.ssh$ ls
id_rsa     id_rsa.pub
[09:20:29]~/.ssh$ scp id_rsa.pub [username]@[hostname]:~/.ssh/authorized_keys
```

you to transfer files between your computer and a remote machine, and works almost exactly like `cp` does. The command has the form `scp [source] [destination]`, where `[source]` (or `[destination]`) can be either a file path on your local computer or the remote machine. When specifying a path on a remote machine, you must include the username and hostname before the path, so it will take the form `[username]@[hostname]:[full-path]`. It is also worth noting that this command should be executed in a local shell session (otherwise you would have to specify a hostname for your own computer, which you may not know). Table **??** reviews several commands for interfacing and transferring data with remote machines.

Table A.2: Commands for interacting with a remote server.

Command	op	Description
`ssh [ops] [username]@[server]`		Login to [server] with account [username]
	`-Y or -X`	Display server output windows locally
1 `scp [ops] [local] [username]@[server]`		Copy file at [local] onto [server]
`scp [ops] [username]@[server] [local]`		Copy file on [server] into [local]
`wget [link]`		Download file from [link]
	`-o [path]`	Save downloaded file at [path]

A.5 Customization & login scripts

Once you have become familiar with navigating and editing through the command line, you can start to consider customizing your environment. Every time you start a bash session, there are two files that are loaded automatically: *.bashrc* and *.bash_profile*. These files set up the bash environment and variables for every new session. By editing these files, users can customize this environment, as well as define their own functions and aliases. In bash, an **aliases** are used to create shorthands for much longer commands. These are a very useful tool for simplifying the commands you use the most. For example, if you frequently login remotely to another machine using the `ssh` command, you can create an alias for this procedure. Instead of having to type `ssh -Y username@serverX.some.domain.name` every single time, you could instead use the alias `serverX`, greatly reducing the amount you need to type when using this command. To do this, you would just need to enter the line `alias serverX='ssh -Y username@serverX.some.domain.name'` into the .bashrc file. It is worth noting that the .bash_profile and .bashrc files are only loaded whenever a new bash session is started, so any changes will not take effect until you logout and log back in again. You can get around this by using the `source` command, which will execute the commands in the file specified. For example, if you have made changes to *.bashrc*, you can give the command `source ~/.bashrc`, and the changes will be loaded without having to re-login.