Big Data in Economics

Lecture 3: Learning to love the shell

Grant McDermott
University of Oregon | EC 510

Table of contents

Prologue

Checklist

☑ Have you cloned the course lecture repo to your local machine?

☑ Once that's done, pull to get the latest lecture slides.

☑ Do you have Bash-compatible shell? (**Windows** users: see here before continuing.)

I'm also going to recommend that you spruce up your GitHub profiles.

- Add your full name, a profile picture, link to website, etc.
- No-one wants to work with (or hire) a barcode.

Today's lecture is the last detour before we get back to data analyis with R and RStudio.

- Laying proper foundations with Git and the shell will put us in a strong positon for advanced data science work as the course develops.

Introduction

The Unix philosophy

The shell tools that we're going to be using today have their roots in the Unix family of operating systems originally developed at Bells Labs in the 1970s.

Besides paying homage, acknowledging the Unix lineage is important because these tools still embody the "Unix philosophy":

> **Do One Thing And Do It Well**

By pairing and chaining well-designed individual components, we can build powerful and much more complex larger systems.

- You can see why the Unix philosophy is also referred to as "minimalist and modular".

Again, this philosophy is very clearly expressed in the design and functionality of the Unix shell.

Definitions

Don't be thrown off by terminology: *shell*, *terminal*, *tty*, *command prompt*, etc.

- These are all basically just different names for the same thing.[1]
- They are all referring to a **command line interface** (CLI).

There are many shell variants, but we're going to focus on **Bash** (i.e. **B**ourne **a**gain **sh**ell).

- Default shell on Linux and MacOS.
- Windows users need to install a Bash-compatible shell first (again, see here).

(For the record, I primarily use Zsh (i.e. Z shell), which is why my shell might look slightly different to yours during live coding sessions. The commands will stay the same, though.)

[1] Truth be told, there are some subtle and sometimes important differences, as well as some interesting history behind the names. But we can safely ignore these here.

Why bother with the shell?

1. Power
    - Both for executing commands and for fixing problems. There are some things you just can't do in an IDE or GUI.
    - It also avoids memory complications associated with certain applications and/or IDEs. We'll get to this issue later in the course.
2. Reproducibility
    - Scripting is reproducible, while clicking is not.
3. Interacting with servers and super computers
    - The shell is often the only game in town for high performance computing. We'll get to this later in the course.
4. Automating workflow and analysis pipelines
    - Easily track and reproduce an entire project (e.g. use a Makefile to combine multiple programs, scripts, etc.)

We're going to focus on 1, 2 and 3 in this course. That's not to say that 4 is unimportant (far from it!), but we just won't have time to cover it.

- Here, here, and here are great places to start learning about automation on your own.

Things that I use the shell for

- Git
- Renaming and moving files *en masse*
- Finding things on my computer
- Combining and manipulating PDFs
- Installing and updating software
- Scheduling tasks
- Monitoring system resources
- Connecting to cloud environments
- Running analyses ("jobs") on super computers
- etc.

Personal aside: One of the (many) nice things about being a Linux user is that it demystifies the shell. You end up using the shell for various day-to-day operations and so inevitably become very comfortable using it. Every year I manage to convince at least one student to switch...

- *I recommend Endeavour OS, though Ubuntu and Elementary are good places to start too.*

Bash shell basics

First look

Let's open up our Bash shell.

A convenient way to do this is through RStudio's built-in Terminal.

- Hitting `Shift` + `Alt` + `T` will cause a "Terminal" tab to open up in the bottom-left window pane (i.e. next to the "Console" tab).
- This should run Bash by default if it is installed on your system. (Windows users: Once again, see here.)

P.S. Of course, it's always possible to open up the Bash shell directly if you prefer.

- Linux
- Mac
- Windows

First look (cont.)

You should see something like:

```
username@hostname:~$
```

This is shell-speak for: "Who am I and where am I?"

- `username` denotes a specific user (one of potentially many on this computer).
- `@hostname` denotes the name of the computer or server.
- `:~` denotes the directory path (where `~` signifies the user's home directory).
- `$` denotes the start of the command prompt.
    - We'll get to this later, but for a special "superuser" called root, the dollar sign will change to a `#`.

Useful keyboard shortcuts

- `Tab` completion.
- Use the `↑` (and `↓`) keys to scroll through previous commands.
- `Ctrl` + `→` (and `Ctrl` + `←`) to skip whole words at a time.
- `Ctrl` + `a` moves the cursor to the beginning of the line.
- `Ctrl` + `e` moves the cursor to the end of the line.
- `Ctrl` + `k` deletes everything to the right of the cursor.
- `Ctrl` + `u` deletes everything to the left of the cursor.
- `Ctrl` + `Shift` + `c` to copy and `Ctrl` + `Shift` + `v` to paste.
- `clear` to clear your terminal.

Syntax

All Bash commands have the same basic syntax:

command option(s) argument(s)

Examples:

```
$ ls -lh ~/Documents/
```

```
$ sort -u myfile.txt
```

Syntax

All Bash commands have the same basic syntax:

command option(s) argument(s)

Examples:

```
$ ls -lh ~/Documents/
```

```
$ sort -u myfile.txt
```

commands

- You don't always need options or arguments. (E.g. `$ ls ~/Documents/` and `$ ls -lh` are both valid commands that will yield output.)
- However, you always need a command.

Syntax (cont.)

options (also called **flags**)

- Start with a dash.
- Usually one letter.
- Multiple options can be chained together under a single dash.

```
$ ls -l -a -h /var/log ## This works
$ ls -lah /var/log ## So does this
```

- An exception is with (rarer) options requiring two dashes.

```
$ ls --group-directories-first --human-readable /var/log
```

arguments

- Tell the command *what* to operate on.
- Usually a file, path, or a set of files and folders.

Help: man

The `man` command ("manual pages") is your friend if you ever need help.

- Tip: Hit spacebar to scroll down a page at a time, "h" to see the help notes of the `man` command itself and "q" to quit.

```
$ man ls
```

```
## LS(1)                          User Commands                          LS(1)
##
## NAME
##        ls - list directory contents
##
## SYNOPSIS
##        ls [OPTION]... [FILE]...
##
## DESCRIPTION
##        List  information  about  the FILEs (the current directory by default).
##        Sort entries alphabetically if none of -cftuvSUX nor --sort  is  speci-
##        fied.
##
##        Mandatory  arguments  to  long  options are mandatory for short options
##        too.
##
```

Help: man (cont.)

A useful feature of `man` is quick pattern searching with "/pattern".

- Try this now by running `$ man ls` again and then typing "/human" (and hitting the return key).

Again, this and other `man` tricks tricks are detailed in the help pages (hit "h").

Help: cheat

I also like the cheat utility, which provides a more readable summary / cheatsheet of various command. You'll need to install it first. (Linux and MacOS only.)

```
$ cheat ls
```

```
## # Displays everything in the target directory
## ls path/to/the/target/directory
##
## # Displays everything including hidden files
## ls -a
##
## # Displays all files, along with the size (with unit suffixes) and timestamp
## ls -lh
##
## # Display files, sorted by size
## ls -S
##
## # Display directories only
## ls -d */
##
## # Display directories only, include hidden
## ls -d .*/ */
```

Files and directories

Navigation

Key navigation commands:

- `pwd` to print (the current) working directory.
- `cd` to change directory.

```
$ pwd
```

```
## /home/grant/Documents/Teaching/EC410-510/lectures/03-shell
```

You can use absolute paths, but it's better to use relative paths and invoke special symbols for a user's home folder (`~`), current directory (`.`), and parent directory (`..`) as needed.

```
$ cd examples ## Move into the "examples" sub-directory of this lecture directory.
$ cd ../.. ## Now go back up two directories.
$ pwd
```

```
## /home/grant/Documents/Teaching/EC410-510/lectures
```

Navigation (cont.)

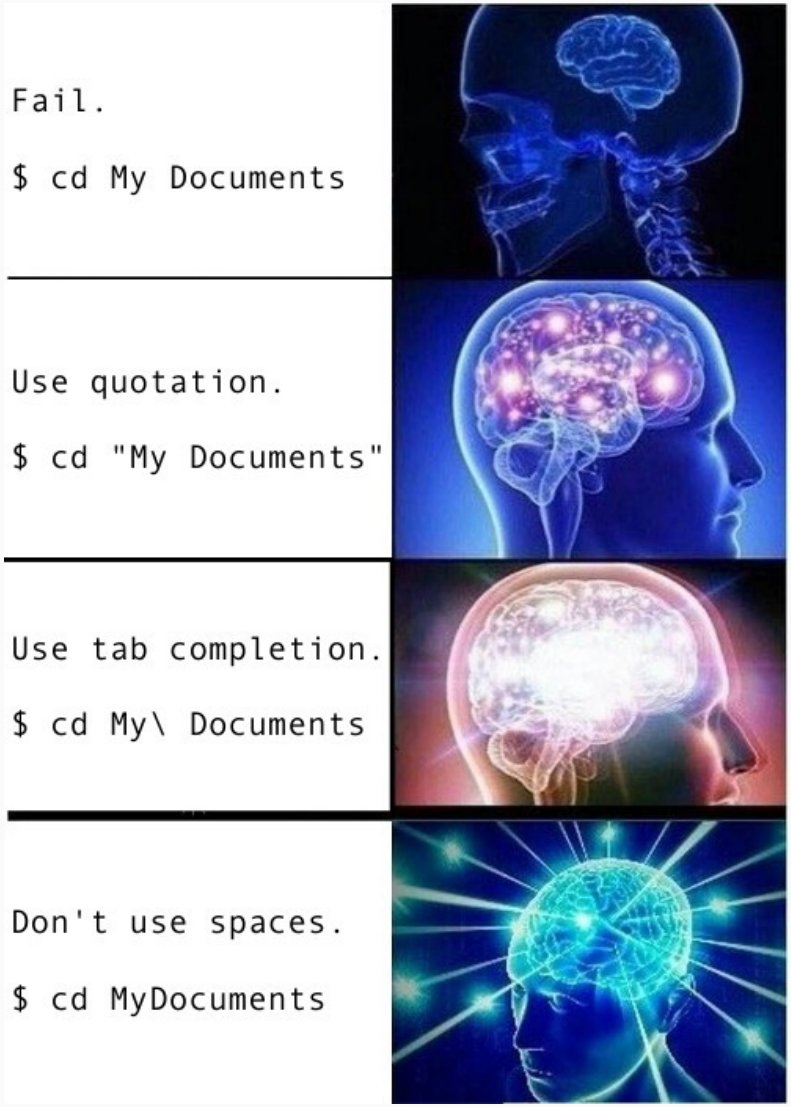
Beware of directory names that contain spaces. Say you have a directory called "My Documents". (I'm looking at you, Windows.)

- Why won't `$ cd My Documents` work?

Answer: Bash syntax is super pedantic about spaces and ordering. Here it thinks that "My" and "Documents" are separate arguments.

- Small brain: Use quotation marks: `$ cd "My Documents"`.
- Big brain: Use Tab completion to automatically "escape" the space: `$ cd My\ Documents`.
- Galaxy brain: Don't use spaces in file and folder names.

Navigation (cont.)



Listing files and their properties

We're about to go into more depth about the `ls` command.

This will be much easier to follow if we're all working with same group of files and folders.

- Navigate to the "examples" sub-directory bundled together with these lecture notes. If you're working through RStudio's terminal, the following should work:

```
$ cd 03-shell/examples ## change relative path as/if needed
```

Now list the directory contents, using the `-lh` flags ("long format", "human readable").

```
$ ls -lh
```

```
## total 152K
## drwxr-xr-x 2 grant users 4.0K Apr 10 12:27 ABC
## drwxr-xr-x 2 grant users 4.0K Jan 12  2019 Bad folder name
## drwxr-xr-x 2 grant users 4.0K Apr 10 12:28 copies
## -rwxr-xr-x 1 grant users   38 Apr 13 17:29 hello.sh
## drwxr-xr-x 2 grant users 4.0K Apr 13 17:05 meals
## -rw-r--r-- 1 grant users   32 Apr 10 14:16 nursery.txt
## -rwxrwxrwx 1 grant users  153 Jan 12  2019 reps.txt
## -rw-r--r-- 1 grant users 120K Jan 12  2019 sonnets.txt
```

Listing files and their properties (cont.)

What does this all mean? Let's focus on the top line.

```
drwxr-xr-x 2 grant users 4.0K Jan 12 22:12 ABC
```

- The first column denotes the object type:
  - `d` (directory or folder), `l` (link), or `-` (file)
- Next, we see the permissions associated with the object's three possible user types: 1) owner, 2) the owner's group, and 3) all other users.
  - Permissions reflect `r` (read), `w` (write), or `x` (execute) access.
  - `-` denotes missing permissions for a class of operations.
- The number of hard links to the object.
- We also see the identity of the object's owner and their group.
- Finally, we see some descriptive elements about the object:
  - Size, date and time of creation, and the object name.

Note: We'll return to file permissions and ownership at the end of the lecture.

Create: touch and mkdir

One of the most common shell tasks is object creation (files, directories, etc.)

We use `mkdir` to create directories. E.g. To create a new "testing" directory:

```
$ mkdir testing
```

We use `touch` to create (empty) files. E.g. To add some files to our new directory:

```
$ touch testing/test1.txt testing/test2.txt testing/test3.txt
```

Check that it worked:

```
$ ls testing
```

```
## test1.txt
## test2.txt
## test3.txt
```

Remove: rm and rmdir

Let's delete the objects that we just created. Start with one of the .txt files, by using `rm`.

- We could delete all the files at the same time, but you'll see why I want to keep some.

```
$ rm testing/test1.txt
```

The equivalent command for directories is `rmdir`.

```
$ rmdir testing
```

```
## rmdir: failed to remove 'testing': Directory not empty
```

Uh oh... It won't let us delete the directory while it still has files inside of it. The solution is to use the `rm` command again with the "recursive" (`-r` or `-R`) and "force" (`-f`) options.

- Excluding the `-f` option is safer, but will trigger a confirmation prompt for every file, which I'd rather avoid here.

```
$ rm -rf testing ## Success
```

Copy: cp

The syntax for copying is `$ cp object path/copyname`

- If you don't provide a new name for the copied object, it will just take the old name.
- However, if there is already an object with the same name in the target destination, then you'll have to use `-f` to force an overwrite.

```
$ ## Create new "copies" sub-directory
$ mkdir copies
$ ## Now copy across a file (with a new name)
$ cp reps.txt copies/reps-copy.txt
$ ## Show that we were successful
$ ls copies
```

```
## mkdir: cannot create directory ‘copies’: File exists
## reps2.txt
## reps-copy.txt
```

You can use `cp` to copy directories, although you'll need the `-r` (or `-R`) flag if you want to recursively copy over everything inside of it to.

- Try this by copying over the `meals/` sub-directory to `copies/`.

Move (and rename): mv

The syntax for moving is `$ mv object path/newobjectname`

```
$  ## Move the abc.txt file and show that it worked
$ mv ABC/abc.txt .
$ ls ABC ## empty
```

```
$ ## Move it back again
$ mv abc.txt ABC
$ ls ABC ## not empty
```

```
## abc.txt
```

Note that "moving" an object within the same directory, but with the (newobjectname) option, is effectively the same as renaming it.

```
$  ## Rename reps-copy to reps2 by "moving" it with a new name
$ mv copies/reps-copy.txt copies/reps2.txt
$ ls copies
```

```
## reps2.txt
```

Rename *en masse*: rename

Speaking of renaming, a more convenient way to do this is with `rename`.

- The syntax is `pattern replacement file(s)`

For example, say we want to change the file type (i.e. extension) of a particular file in the `meals` directory.

```
$ rename csv TXT meals/monday.csv
$ ls meals
```

```
## friday.csv
## mealplan.csv
## monday.TXT
## saturday.csv
## sunday.csv
## thursday.csv
## tuesday.csv
## wednesday.csv
```

Note: Mac users must install `rename` from Homebrew (`$ brew rename`). This Mac version also requires slightly different syntax; you'll need to include an `-s` ("substitute") flag in your commands here, e.g. `$ rename -s csv TXT meals/monday.csv`

Rename *en masse*: rename (cont.)

Where `rename` really shines, however, is in conjunction with regular expressions and wildcards (more on the next slide).

- This works especially well for dealing with a whole list of files or folders.

For example, let's change *all* of the file extensions in the `meals` directory.

```
$ rename csv TXT meals/*
$ ls meals
```

```
## friday.TXT
## mealplan.TXT
## monday.TXT
## saturday.TXT
## sunday.TXT
## thursday.TXT
## tuesday.TXT
## wednesday.TXT
```

Better change them back before we continue. (Confirm that this worked for yourself.)

```
$ rename TXT csv meals/*
```

Wildcards

Wildcards are special characters that can be used as a replacement for other characters. The two most important ones are:

1. Replace any number of characters with `*`.
    - Convenient when you want to copy, move, or delete a whole class of files.

```
$ cp *.sh copies ## Copy any file with an .sh extension to "copies"
$ rm copies/* ## Delete everything in the "copies" directory
```

2. Replace a single character with `?`
    - Convenient when you want to discriminate between similarly named files.

```
$ ls meals/??nday.csv
$ ls meals/?onday.csv
```

```
## meals/monday.csv
## meals/sunday.csv
## meals/monday.csv
```

Find

The last command that I want to mention w.r.t. navigation is `find`.

- This can be used to locate files and directories based on a variety of criteria; from pattern matching to object properties.

```
$ find -iname "monday.csv" ## will automatically do recursive
```

```
## ./meals/monday.csv
```

```
$ find . -iname "*.txt" ## must use "." to indicate pwd
```

```
## ./nursery.txt
## ./sonnets.txt
## ./reps.txt
## ./survive.txt
## ./ABC/abc.txt
```

```
$ find . -size +100k ## find files larger than 100 KB
```

```
## ./sonnets.txt
```

Working with text files

Motivation

Economists and other (data) scientists spend a lot of time working with text, including scripts, Markdown documents, and delimited text files like CSVs.

It therefore makes sense to spend a few slides showing off some Bash shell capabilities for working with text files.

- We'll only scratch the surface, but hopefully you'll get an idea of how powerful the shell is in the text domain.

Counting text: wc

You can use the `wc` command to count: 1) lines of text, 2) the number of words, and 3) the number of characters.

Let's demonstrate with a text file containing all of Shakespeare's Sonnets.[1]

```
$ wc sonnets.txt
```

```
##   3029  20701 122780 sonnets.txt
```

PS — You couldn't tell here, but the character count is actually higher than we'd get if we (bothered) counting by hand, because `wc` counts the invisible newline character "\n".

[1] Courtesy of Project Gutenburg.

Reading text

Read everything: cat

The simplest way to read in text is with the `cat` ("concatenate") command. Note that `cat` will read in *all* of the text. You can scroll back up in your shell window, but this can still be a pain.

Again, let's demonstrate using Shakespeare's Sonnets. (This will overflow the slide.)

- I'm also going to use the `-n` flag because I want to show line numbers.

```
$ cat -n sonnets.txt
```

```
##      1    The Project Gutenberg EBook of Shakespeare's Sonnets, by William Shakespea
##      2
##      3    This eBook is for the use of anyone anywhere at no cost and with
##      4    almost no restrictions whatsoever.  You may copy it, give it away or
##      5    re-use it under the terms of the Project Gutenberg License included
##      6    with this eBook or online at www.gutenberg.org
##      7
##      8
##      9    Title: Shakespeare's Sonnets
##     10
```

Reading text (cont.)

Scroll: more and less

The `more` and `less` commands provide extra functionality over `cat`. For example, they allow you to move through long text one page at a time.

- Try this yourself with `$ more sonnets.txt`
- You can move forward and back using the "f" and "b" keys, and quit by hitting "q".

Preview: head and tail

The `head` and `tail` commands let you limit yourself to a preview of the text, down to a specified number of rows. (The default is 10 rows if you don't specify a number.)

```
$ head -n 3 sonnets.txt ## First 3 rows
$ # head sonnets.txt ## First 10 rows (default)
```

```
## The Project Gutenberg EBook of Shakespeare's Sonnets, by William Shakespeare
##
## This eBook is for the use of anyone anywhere at no cost and with
```

Reading text (cont.)

Preview: head and tail (cont.)

`tail` works very similarly to `head`, but starting from the bottom. For example, we can see the very last row of a file as follows

```
$ tail -n 1 sonnets.txt ## Last row
```

```
## subscribe to our email newsletter to hear about new eBooks.
```

However, there's one other neat option that I want to show you. By using the `-n +N` option, we can specify that we want to preview all lines starting from row N *and after*. E.g.

```
$ tail -n +3024 sonnets.txt ## Show everything from line 3024
```

```
##      www.gutenberg.org
##
## This Web site includes information about Project Gutenberg-tm,
## including how to make donations to the Project Gutenberg Literary
## Archive Foundation, how to help produce our new eBooks, and how to
## subscribe to our email newsletter to hear about new eBooks.
```

Find patterns: grep

To find patterns in text, we can use regular expression-type matching with `grep`.

For example, say we want to find the famous opening line to Shakespeare's Sonnet 18.

- I'm going to include the `-n` ("number") flag to get the line that it occurs on.

```
$ grep -n "Shall I compare thee" sonnets.txt
```

```
## 336:  Shall I compare thee to a summer's day?
```

By default, `grep` returns all matching patterns.

- What happens if you run `$ grep -n "summer" sonnets.txt`?
- Or, for that matter, `$ grep -n "the" sonnets.txt`?

Find patterns: grep (cont.)

Note that `grep` can be used to identify patterns in a group files (e.g. within a directory) too.

- This is particularly useful if you are trying to identify a file that contain, say, a function name.

Here's a simple example: Which days will I eat pasta this week?

- I'm using the `R` (recursive) and `l` (just list the files; don't print the output) flags.

```
$ grep -Rl "pasta" meals
```

```
## meals/mealplan.csv
## meals/monday.csv
```

What about muesli? And pizza?

Take a look at the `grep` man or cheat file for other useful examples and flags (e.g. `-i` for ignore case).

PS — Another cool (and very fast) shell utility along these lines is the silver searcher. Check it out.

Manipulate text: sed and awk

There are two main commands for manipulating text in the shell, namely `sed` and `awk`.

- Both of these are very powerful and flexible (`awk` is particularly good with CSVs).

I'm going to show two basic examples without going into depth, but I strongly encourage you to explore more on your own. (Mac users: See here.)

Example 1. Replace one text pattern with another.

```
$ cat nursery.txt
```

```
## Bill and Jill
## Went up the hill
```

Now, change "Jack" to "Bill".

```
$ sed -i 's/Jack/Bill/g' nursery.txt
$ cat nursery.txt
```

```
## Bill and Jill
## Went up the hill
```

Manipulate text: sed and awk (cont.)

Example 2. Find and count the 10 most commonly used words in Shakespeare's Sonnets.

- Note: We'll learn more about the pipe operator (`|`) in a few slides.

```
$ sed -e 's/\s/\n/g' < sonnets.txt | sort | uniq -c | sort -nr | head -10
```

```
##    8884
##     513 the
##     456 of
##     401 to
##     341 my
##     338 in
##     327 I
##     316 and
##     252 thy
##     248 that
```

PS — You can also use double quotes (") instead of single ones (') for `sed` and `awk` commands. This can sometimes run you into trouble with special symbols or patterns in

Sorting and removing duplicates: sort

We can remove duplicate lines in various ways in Bash, but I'll demonstrate using `sort`.

```
$ cat reps.txt
```

```
## Sometimes I repeat myself.
## Sometimes I repeat myself.
## Other times not so much.
## But I try to be good.
## Other times not so much.
## Sometimes I repeat myself.
```

There's a fair bit of repetition in this file (and a double entendre). Let's fix that.

- Note the use of the `-u` ("unique") flag to remove duplicates. I'll also add a `-r` ("reverse") flag, but only because `sort` orders alphabetically and this makes less sense for this simple example.

```
$ sort -ur reps.txt
```

```
## Sometimes I repeat myself.
## Other times not so much.
## But I try to be good.
```

Redirecting and pipes

Redirect: >

You can send output from the shell to a file using the redirect operator `>`

For example, let's print a message to the shell using the `echo` command.

```
$ echo "At first, I was afraid, I was petrified"
```

```
## At first, I was afraid, I was petrified
```

If you wanted to save this output to a file, you need simply redirect it to the filename of choice.

```
$ echo "At first, I was afraid, I was petrified" > survive.txt
$ find survive.txt ## Show that it now exists
```

```
## survive.txt
```

Redirect: > (cont.)

If you want to *append* text to an existing file, then you should use `>>`.

- Using `>` will try to overwrite the existing file contents.

```
$ echo "'Kept thinking I could never live without you by my side" >> survive.txt
$ cat survive.txt
```

```
## At first, I was afraid, I was petrified
## 'Kept thinking I could never live without you by my side
```

(Don't be shy. You can hum the rest of the song to yourself now.)

Aside: I often use this sequence when adding files to my .gitignore. E.g. `$ echo "*.csv" >> .gitignore`.

Pipes: |

The pipe operator `|` is one of the coolest features in Bash.

- It lets you send (i.e. "pipe") intermediate output to another command.
- In other words, it allows us to chain together a sequence of simple operations and thereby implement a more complex operation. (Remember the Unix philosophy!)

Let me demonstrate using a very simple example.

```
$ cat -n sonnets.txt | head -n100 | tail -n10
```

```
##     91      Despite of wrinkles this thy golden time.
##     92        But if thou live, remember'd not to be,
##     93        Die single and thine image dies with thee.
##     94
##     95      IV
##     96
##     97      Unthrifty loveliness, why dost thou spend
##     98      Upon thy self thy beauty's legacy?
##     99      Nature's bequest gives nothing, but doth lend,
##    100      And being frank she lends to those are free:
```

Pipes: | (cont.)

An exercise: Say I want to pull out all of text from (but limited to) Sonnet 18.

- How might you go about this task using the pipe and other Bash commands?
- Tip: Use your knowledge of the starting line (i.e. 336) and the fact that sonnets are 14 lines long.

```
$ tail -n +336 sonnets.txt | head -n14
```

```
##   Shall I compare thee to a summer's day?
##   Thou art more lovely and more temperate:
##   Rough winds do shake the darling buds of May,
##   And summer's lease hath all too short a date:
##   Sometime too hot the eye of heaven shines,
##   And often is his gold complexion dimm'd,
##   And every fair from fair sometime declines,
##   By chance, or nature's changing course untrimm'd:
##   But thy eternal summer shall not fade,
##   Nor lose possession of that fair thou ow'st,
##   Nor shall death brag thou wander'st in his shade,
##   When in eternal lines to time thou grow'st,
##     So long as men can breathe, or eyes can see,
##     So long lives this, and this gives life to thee.
```

Pipes: | (cont.)

A final aside about pipe the friends: You can use it to search through your Bash command history.

- Every shell command you type is stored in a `~/.bash_history` file.[1]

What happens if you type `$ cat ~/.bash_history | grep head`?

FWIW, I use this approach often to remind myself of certain shell commands that I tend to forget.

[1] The file might change depending on your preferred shell. E.g. For Zsh it's `~/.zhistory`.

Iteration (*for* loops)

for loop syntax

for loops in Bash work similarly to other programming languages that you are probably familiar with.

The basic syntax is

```
for i in LIST
do
  OPERATION $i ## the $ sign indicates a variable in bash
done
```

We can also condense things into a single line by using ";" appropriately.

```
for i in LIST; do OPERATION $i; done
```

I find the top approach more readable, but I may use single line approach in these slides to save vertical space.

- Note: Using ";" isn't limited to *for loops*. Semicolons are a standard way to denote line endings in Bash.

Example 1: Print a sequence of numbers

To help make things concrete, here's a simple *for* loop in action.

```
$ for i in 1 2 3 4 5; do echo $i; done
```

```
## 1
## 2
## 3
## 4
## 5
```

FWIW, we can use bash's brace expansion (`{1..n}`) to save us from having to write out a long sequence of numbers.

```
$ for i in {1..5}; do echo $i; done
```

```
## 1
## 2
## 3
## 4
## 5
```

Example 2: Combine CSVs

Here's a more realistic *for* loop use-case that I use quite often: Combining (i.e. concatenating) multiple CSVs.

Say we want to combine all the "daily" files in the `meals` directory into a single CSV, which I'll call `mealplan.csv`. Here's one attempt that incorporates various bash commands and tricks that we've learned so far. The basic idea is:

1. Create a new (empty) CSV
2. Then, loop over the relevant input files, appending their contents to our new CSV

```
## create an empty CSV
$ touch meals/mealplan.csv
## loop over the input files and append their contents to our new CSV
$ for i in $(ls meals/*day.csv)
> do
>   cat $i >> meals/mealplan.csv
> done
```

Did it work? (See next slide.)

Example 2: Combine CSVs (cont.)

Hmmm. Sort of, but we need to get rid of the repeating header.

```
$ cat meals/mealplan.csv
```

```
## day,breakfast,lunch,dinner
## friday,pancakes,ramen,stew
## monday,muesli,sandwich,pasta
## saturday,muesli,sandwich,pad thai
## sunday,muesli,roast,leftovers
## thursday,muesli,salad,tacos
## tuesday,muesli,soup,roast
## wednesday,muesli,sandwich,pizza
## day,breakfast,lunch,dinner
## friday,pancakes,ramen,stew
## day,breakfast,lunch,dinner
## monday,muesli,sandwich,pasta
## day,breakfast,lunch,dinner
## saturday,muesli,sandwich,pad thai
## day,breakfast,lunch,dinner
## sunday,muesli,roast,leftovers
## day,breakfast,lunch,dinner
## thursday,muesli,salad,tacos
## day,breakfast,lunch,dinner
```

Example 2: Combine CSVs (cont.)

Let's try again. First delete the old file so we can start afresh.

```
$ rm -f meals/mealplan.csv ## delete old file
```

Here's our adapted gameplan:

- First, create the new file by grabbing the header (i.e. top line) from any of the input files and redirecting it. No need for `touch` this time.
- Next, loop over all the input files as before, but this time only append everything *after* the top line.

```
## create a new CSV by redirecting the top line of any file
$ head -1 meals/monday.csv > meals/mealplan.csv
## loop over the input files, appending everything after the top line
$ for i in $(ls meals/*day.csv)
> do
>   tail -n +2 $i >> meals/mealplan.csv
> done
```

Example 2: Combine CSVs (cont.)

It worked!

```
$ cat meals/mealplan.csv
```

```
## day,breakfast,lunch,dinner
## friday,pancakes,ramen,stew
## monday,muesli,sandwich,pasta
## saturday,muesli,sandwich,pad thai
## sunday,muesli,roast,leftovers
## thursday,muesli,salad,tacos
## tuesday,muesli,soup,roast
## wednesday,muesli,sandwich,pizza
```

We still have to sort the correct week order, but that's an easy job in R (or Stata, Python, etc.)

- The explicit benefit of doing the concatenating in the shell is it is *much* more efficient, since all the files don't simultaneously have to be held in memory (i.e RAM).
- This doesn't matter here, but can make a dramatic difference once we start working with lots of files (or even a few really big ones). We'll revisit this idea later in the big data section of the course.

Scripting

Hello World!

Writing code and commands interactively in the shell makes a lot of sense when you are exploring data, file structures, etc.

However, it's also possible (and often desirable) to write reproducible shell scripts that combine a sequence of commands.

- These scripts are demarcated by their `.sh` file extension.

Let's look at the contents of a short script that I've included in the examples folder.

```
$ cat hello.sh
```

```
## #!/bin/sh
## echo -e "\nHello, World!\n"
```

I'm sure that you already have a good idea of what this script is meant to do, but it will prove useful to quickly go through some things together.

Hello World! (cont.)

```
#!/bin/sh
echo -e "\nHello World!\n"
```

- `#!/bin/sh` is a shebang, indicating which program to run the command with (here: any Bash-compatible shell). However, it is typically ignored (note that it begins with the hash comment character.)
- `echo -e "\nHello World!\n"` is the actual command that we want to run.

To run this simple script, you can just type in the relative path of and press enter. (The relative path is needed, otherwise bash thinks `hello.sh` is an internal command.)

```
$ ./hello.sh
$ # bash hello.sh ## Also works (and doesn't need leading "./")
```

```
##
## Hello, World!
```

Editing and writing with nano

Say you want to edit my (amazing) script.

- Maybe you want to add some additional lines of text, or maybe you're bothered by the fact that there should be a comma after "Hello". (It's a salutation, dammit!)

We have already seen how to append text lines to a file, but when it comes to larger jobs or more complicated editing work, you're better off using nano.

- In-built Unix shell text editor. (**Windows users**, see here.)

Open up the script in nano by typing `$ nano hello.sh`.

- Note that the functionality is more limited than a normal text editor.
- Once you are finished editing, hit "Ctrl+X", then "y" and enter to exit.
- Finally, run the edited version of the script.

If you've been having trouble executing this script (or want to limit who else can execute it), then you need to alter its permissions. Which takes us neatly on to our final section...

User roles and file permissions

Disclaimer

This next section is tailored towards Unix-based operating systems, like Linux or MacOS, which is why I have saved it for the end. ***Windows users:*** *Don't be suprised if some commands don't work, especially if you haven't installed the WSL...*

Regardless, the things we learn here will become relevant to everyone (even Windows users) once we start interacting with Linux servers, spinning up Docker containers, etc. later in the course.

The superuser: root, sudo, etc.

There are two main user roles on a Linux system:

1. Normal users
2. A superuser (AKA "root")

Difference is one of priviledge.

- Superusers can make system changes, install software, browse through different users' home folders, etc. Normal users are much more restricted in what they can do.
- Explains why Unix-based OS's are much more resilient to security threats like viruses. Need superuser priviledges to install (potentially malicious) software.

The superuser: root, sudo, etc. (cont.)

You *can* log in as the superuser[1]... but this is generally considered very poor practice, since you needlessly risk messing up your system.

- There are no safety checks and no "undo" options.

Question: How, then, can normal users perform meaningful system operations (including installing new programs and updating software)?

Answer: Invoke temporary superuser status with `sudo`.

- Stands for "superuser do".
- Simply prepend `sudo` to whatever command you want to run.

```
grant@laptop:~$ ls /root ## fails
grant@laptop:~$ sudo ls /root ## works
```

[1] Hit "p" on this slide if you really want to know how.

Changing permissions and ownership

Let's think back to the `ABC/` directory that we saw previously while exploring the ls command.

```
drwxr-xr-x 2 grant users 4.0K Jan 12 22:12  ABC
```

We can change the permissions and ownership of this folder with the `chmod` and `chown` commands, respectively. We'll now review these in turn.

- Note that I'm going to use the "recursive" option (i.e. `-R`) in the examples that follow, but only because `ABC/` is a directory. You can drop that when modifying individual files.

chmod

Changing permissions using `chmod` depends on how those permissions are represented.

There are two options: 1) Octal notation and 2) Symbolic notation.

- We'll go into more detail on the next slide, but let's see some examples first.
- (Test the results yourself using the `ls -lh` command afterwards.)

Example 1: rwxrwxrwx. Read, write and execute permission for all users.

- Octal: `$ chmod -R 777 ABC`
- Symbolic: `$ chmod -R a=rwx ABC`

Example 2: rwxr-xr-x. Read, write and execute permission for the main user (i.e. owner) of the file. For all other users, read and execute permission only.

- Octal: `$ chmod -R 755 ABC`
- Symbolic: `$ chmod -R u=rwx,g=rx,o=rx ABC`

Now that we've seen some examples, let's get into the logic behind them.

chmod (cont.)

Octal notation

Takes advantage of the fact that `4` (for "read"), `2` (for "write"), and `1` (for "execute") can be combined in unambiguous ways.

- 7 (= 4 + 2 + 1) means read, write and execute permission.
- 5 (= 4 + 0 + 1) means read and execute permission, but not write permission.
- etc.
- Note that Octal notation requires a number for each of the three user types: owner, owner's group, and all others. E.g. `$ chmod 777 myfile.txt`

Symbolic notation

Links permissions to different symbols (i.e. abbreviations).

- Users: `u` ("User/owner"), `g` ("Group"), `o` ("Others""), `a` ("All")
- Permissions: `r` ("read"), `w` ("write"), `x` ("execute")
- Changes: `+` ("add permissions"), `-` ("remove permissions"), `=` ("set new permissions")

chmod (cont.)

Here's a quick comparison table with some common permission levels.

Octal value	Symbolic value	Permission level
777	a+rwx	rwxrwxrwx
770	u+rwx,g+rwx,o-rwx	rwxrwx---
755	a+rwx,g=rw,o=rw	rwxr-xr-x
700	u+rwx,g-rwx,o-rwx	rwx------
644	u=rw,g=r,o=r	rw-r--r--

PS — Note the Symbolic method allows for relative changes, which means that you don't necessarily need to write out the whole entry in the table above. E.g. To go from the first line to the second line, you'd only need `$ chmod o-rwx myfile`.

chown

Changing file ownership is somewhat easier than changing permissions, because you don't have to remember the different Octal and Symbolic notation mappings.

- E.g. Say there is another user on your computer called "alice", then you could just assign her ownership of the ABC subfolder using:

```
$ chown -R alice ABC
```

Things get a little more interesting when we want to add new users and groups, or change an existing users group.

- I'll save that for a later lecture on cloud servers, though.

Next steps

Things we didn't cover today

I know we covered a *lot* of ground today. I hope that I've given you a sense of how Bash works and how powerful it is.

- My main goal has been to "demystify" the shell, so that you aren't intimidated when we use shell commands later on.

At the same time, there's loads that we didn't cover.

- Environment variables, SSH, memory management (e.g. top and htop), automating tasks (e.g. cron jobs), GNU parallel, etc.
- We'll get to some of these topics in the later lectures, but please try to work some of the suggested exercises on the next slide and make use of the recommended readings.

Next steps

Exercises

I want you to continue playing around with some of the different Bash commands that we practiced today.

- Change the permissions on an individual file or a whole directory.
- Read in (or fix) some lines of text from one file and pipe them to another file.
- Count the number of times Shakespeare refers to "mistress" or "love" in his Sonnets.
- Write a new bash script and execute it.
- Etc.

Further reading

- The Unix Shell (Software Carpentery)
- The Unix Workbench (Sean Kross)
- Data Science at the Command Line (Jeroen Janssens)
- Using AWK and R to parse 25tb (Nick Strayer)

Next class: *R* language basics

Appendix (Windows users only)

Bash on Windows

Windows users have two options:

1. Git Bash

- **Pros:** You should already have installed this as part of the previous lecture.
- **Cons:** Functionality is limited to Git-related commands, so various things that we're going to practice today won't work.

2. Windows Subsystem for Linux (WSL)

- **Pros:** A self-contained Linux image (terminal) that allows full Bash functionality.
- **Cons:** Must be installed first and only available to Windows 10 users.

I'm going to go out on a limb and recommend option 2 (WSL) if available to you. It's more overhead, but I think worth it. See the next two slides for instructions and tips...

(Back to table of contents.)

WSL

The basic WSL installation guide is **here**.

- Follow the guide to install your preferred Linux distro. Ubuntu is a good choice.
- Then, once you've restarted your PC, come back to these slides.

After installing your chosen WSL, you need to navigate to today's lecture directory to run the examples. You have two options:

i) Go directly through the WSL

Presumably, you've cloned the course repo somewhere on your C drive.

- The way this works is that Windows drives are mounted on your WSL's `mnt` directory.
- Say you cloned the repo to "C:\Users\Grant\ec510\lectures".
- Then, the WSL equivalent is "/mnt/c/Users/Grant/ec510/lectures".
- So, you can navigate to today's lecture directory through your WSL with: `$ cd /mnt/c/Users/Grant/ec510/lectures/03-shell`. Adjust as needed.

Continues on next slide...

WSL (cont.)

ii) Access WSL through RStudio

If you access WSL through RStudio, then it will conveniently configure your path to the present working directory. So, here's how to make WSL your default RStudio Terminal:

- In RStudio, navigate to: *Tools* > *Terminal* > *Terminal Options*.... [Screenshot.]
- Click on the dropdown menu for *New terminals open with* and select "Bash (Windows Subsystem for Linux)", Then click *OK*. [Screenshot.]
- Refresh your RStudio terminal (`Alt+Shift+R`). [Screenshot.]
- You should see the WSL Bash environment with the path automatically configured to the present working director, mount point and all. [Screenshot.]

Which option to choose?

Both are fine, but I recommend option **(ii)**. As a Windows user, being able to access a true Bash shell (i.e. terminal) conveniently from RStudio will make things *much* easier for you in my class. You can always revert back to a different shell later if you want.

(Back to table of contents.)