Version control and the command line

Tad Dallas

Unix (bash) shell

The best way to directly interact with your computer is through the terminal window. This terminal (in Mac and Linux OSs) is the unix shell, and is commonly referred to as the bash shell, though bash is only one flavor of unix shell (e.g., Ksh shell, fish).

We will use the bash shell to run some of our programs, and help with makefiles and travis-CI builds, which we will go over at some point. On a basic level, the ability to us the bash shell is incredibly useful for copying/moving/deleting files (especially in bulk) and the automation of repetitive tasks.

Tutorial

Open up the terminal.

You should see, at least in Ubuntu, your username followed by a "@" and the name of your machine (e.g., `tad@poe`), followed by ":~$" and a blinking cursor. You are in the home directory by default. To see this, issue the command

```
pwd
```

which should output "/home/name", where "name" is your username (again, this may be specific to Linux OSs).

To change the directory that you are in, you use the `cd` command.

```
cd Documents
```

should navigate you to the Documents folder (assuming that it exists). To test if it exists, we can examine all the items in the current directory by using the `ls` command.

```
ls -l
```

Here, we issued the `ls` command with the `-l` argument. Adding arguments to functions allows for customizable output. Here, we ask ls to output additional information on modification date, permissions, file size, etc. To see the other possible arguments we could have given the function, we can use

```
ls --help
man ls
```

The `--help` argument should work with any shell based command.

Directories are structured hierarchically, so `home/tad/Documents` means that the `Documents` folder is nested within the `tad` folder which is nested within `home`. To back up levels, we can use relative paths. For instance, if we wish to back up one level,

```
cd '..'
```

or two levels

```
cd '../..'
```

By default, if you issue `cd` without any arguments, you will be returned to your home directory.

Wildcards Another useful tool is the use of wildcards, which allow you to target certain files without knowing their exact name. For instance, if you want to list all files in the working directory that have a certain extension (e.g., '.txt'), you could issue the command

```
ls *.txt
```

Here, the star means "anything" and the .txt constrains the ls to just .txt files.

A few other important functions are given below. The best way to learn about them is to read the –help file, and to play around with using them. I list the commands below, and then we can take some time to practice interacting with our machines through bash shell.

- cp: copy a file from one location to another
- mv: move a file from one location to another
- rm: remove a file (permanent)
- mkdir: create a new folder
- touch: create a file
- cat: display content of a file
- head: display first 10 lines of a file
- tail: display last 10 lines of a file
- wc: count lines, words, and characters in a file
- echo: print text to console
- cat: print text to console, but upset a subset of people
- nano: command-line text editor that ships with Ubuntu
- top: show the usage stats

Practice

Do all of this from command line. 1. create a folder called "firstRepo" 2. navigate into the folder 3. create a file called "README.md" 4. use nano to write "This is my first GitHub repo", save and exit.

Downloading programs that you need

This section will focus on how to install things on Ubuntu. If we want to install a program called htop, we could issue:

```
apt-get install htop
```

However, this should error out and saying something like "Permission denied" and ask "are you root?". You are not. Being "root" means you have permissions to install and modify all files. We will not go into permissions in much detail here. However, to get root access, you have to pre-empt the command by "sudo", and you will be prompted for your password.

```
sudo apt-get install htop
sudo apt install htop
```

You also do not strictly need the "-get" part of that. This is a holdover from an earlier version, and I just stuck with it due to muscle memory.

Finally, this should rarely (fingers crossed) be the case when working on these lab computers for people in the course, as I requested that all (hopefully) necessary packages and such be installed prior to you sitting in that seat.

---

git

Now we will demonstrate how to use command line utilities. These are things that do not have a GUI (graphical user interface), and therefore must be run through the command line (e.g., nano). The thing we will learn to use through command line is a version control software called `git`.

What is version control?

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

Why does it matter?

Version control is important both for collaborative coding with team members, and for developing your own code locally. Using version control software as an individual is important to have a constant backup of previously developed files, charting a clear timeline of progress on a particular project. Using version control software as a team helps keep every team member working off the latest version, or allows team members to develop add-ons without breaking the main flow of the code.

How do I do it?

I want you to use the folder you made above called `firstRepo` as your first GitHub repo. So navigate to it from command line. Now, initiate the folder as a git repository (i.e., repo).

```
git init
```

This creates a `.git` folder which houses information on each version of the repo. A quick aside that we will have to do is

```
git remote add origin '/path/to/your/folder'
```

This part will become unnecessary when we interface with Github (discussed below), and is only needed when you are setting up a version controlled project that will only ever exist on your local machine. There are still numerous benefits to version controlling your work on your own machine, but the power of git goes well beyond this.

To properly version control, you have to take periodic "snapshots" of the contents of the repo, tracking changes in files across time. To take a "snapshot", we have to do a "commit". This is `git` jargon, and I will go over it below, introducing all the terms at once to hopefully provide a glossary and clear programmatic flow.

```
git add
git commit -m 'message'
git push origin master
```

So all git commands start by using the prefix "git", as above, into the bash shell. The order of events is as such. First, we `add` files to the `commit`, telling git which files we want to include in the commit. Files that we do not include are not removed, but they just are not versioned in that commit.

Next, we make a `commit`, and supply some informative message about what was changed using the `-m` argument to the `commit` function.

After this, we simply push the changes, which takes all the changes that were staged during the process of adding the files and then committing the files, and creates a clear record.

The use of git with Github

Github is an online hosting platform for your git repositories. That is, you can maintain a versioned history of your files independent of the internet and any potential collaborators, but by hosting on a platform like Github, you can collaboratively develop files with other people, and everything remains nicely versioned.

This is important, as this will be how you turn in assignments. I expect that you will leave this class with a solid working knowledge of git and Github.

Setting up a Github account

Go to https://github.com/ and set up your account if you have not already done so.

Making your local git repo talk to Github

Recall when we added a `remote` to the local git repo? To refresh your memory, we issued this command

```
git remote add origin '/path/to/your/folder'
```

which was just a workaround that we probably did not even strictly need. The reason why we did not need it is because `remotes`, by definition, are for remote projects. That is, we use `remotes` to manage projects that will be hosted on internet services (like Github!). So now we will change the remote of this project to point to Github. To do this, we first have to set up the remote repo on Github.

To do this, we will navigate to our account on Github, and click the `+` dropdown menu in the top right corner, selecting "New repository". We fill out the relevant information, and create it. Then, we can go back to our local repo, and set the remote to point to Github.

```
git remote add origin https://github.com/userName/repoName.git
```

Now, when we go through the `add, commit, push` process of staging and making a commit (as described above), you will be prompted for your Github username and password.

you may also need to set some global options of your name and email address when first pushing to Github, but you will be instructed on how exactly to do this

Benefits of Github

Github allows for collaborative coding, meaning that people distributed across the world can work on different aspects of incredibly complicated things, including entire languages (e.g., Rust, Julia, etc.), machine learning frameworks (e.g., Tensorflow), and a large collection of operating systems (e.g., https://github.com/jubalh/awesome-os).

What does this mean for you?

Probably nothing, but it means something for how you will collaborate with your classmates, and how you will turn in assignments. All your dev work for your assignments will be version controlled on Github.

So let's take a tour of Github

this will be done in class

sessionInfo

```
sessionInfo()
```

```
## R version 4.3.1 (2023-06-16)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 22.04.2 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.10.0
## LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.0
##
## locale:
```

```
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## time zone: America/New_York
## tzcode source: system (glibc)
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] dplyr_1.1.2    plyr_1.8.8     DBI_1.1.3      rgbif_3.7.7    jsonlite_1.8.7
## [6] httr_1.4.6     rmarkdown_2.23
##
## loaded via a namespace (and not attached):
##  [1] utf8_1.2.3        generics_0.1.3    xml2_1.3.5        RSQLite_2.3.1
##  [5] stringi_1.7.12    httpcode_0.3.0    digest_0.6.33     magrittr_2.0.3
##  [9] evaluate_0.21     grid_4.3.1        fastmap_1.1.1     blob_1.2.4
## [13] maps_3.4.1        whisker_0.4.1     crul_1.4.0        tinytex_0.45
## [17] urltools_1.7.3    purrr_1.0.1       fansi_1.0.4       scales_1.2.1
## [21] oai_0.4.0         lazyeval_0.2.2    cli_3.6.1         rlang_1.1.1
## [25] dbplyr_2.3.2      triebeard_0.4.1   bit64_4.0.5       munsell_0.5.0
## [29] withr_2.5.0       cachem_1.0.8      yaml_2.3.7        tools_4.3.1
## [33] memoise_2.0.1     colorspace_2.1-0  ggplot2_3.4.2     curl_5.0.1
## [37] vctrs_0.6.3       R6_2.5.1          lifecycle_1.0.3   stringr_1.5.0
## [41] bit_4.0.5         pkgconfig_2.0.3   pillar_1.9.0      gtable_0.3.3
## [45] data.table_1.14.8 glue_1.6.2        Rcpp_1.0.11       xfun_0.39
## [49] tibble_3.2.1      tidyselect_1.2.0  highr_0.10        knitr_1.43
## [53] htmltools_0.5.5   compiler_4.3.1
```