Command Line Tools

Luke Motley

University of Chicago

October 2024

Why should I use the command line?

Things the command line can do in a pinch

- Find and count the 10 most common words in all files called `evidence.txt`.
- Find “Luke” in files called `evidence.txt` and replace him with “Amedeus”.
- Find every instance of `evidence.txt` on your computer and destroy it.

Why should I use the command line?

'The command line is our means of implementing tools.'

Gentzkow and Shapiro (RA Manual)

↑ They are good at research.

Why should I use the command line?

- Efficiency
- Automation
- Reproducibility
- You look like a hacker.

Compared to what?

We edit text and run code for a living.

The former can be done in text editors (e.g., VSCode).

The latter can be done in GUIs (e.g., RStudio).

The command line can do both.

Roadmap

Command line 101

GitHub, the right way

Automate what can be automated

Living in the terminal

Command line 101

What is the command line?

The interface that allows you to perform actions on your computer with commands.

- Basically synonyms[1]: *terminal*, *shell*, *command line*, *CLI*, *command prompt*.

[1] I suspect this is pragmatically true for economists. Personally, I am not good enough to need to know the difference between them. I open my *terminal*/*command line*/*shell* and do something.

Disclaimer

I am on a Mac and use *zsh* shell.

The slides will refer to *bash* shell, since it is more common.

The basic syntax identical, and the commands themselves are almost the same (disclaimer: there are headache-inducing edge cases).

If you're on Windows, you will not have a *bash* shell by default, but you can install one.

I'll also use this slide to mention that I have only recently come to use all of this stuff into my workflow. I am definitely not qualified to teach this. But these tools have totally changed the way to approach the job, so hopefully I can at least convince you to give them a try.

Command to English

At the beginning, you will forget a lot of these commands and have to Google how to do things.

A silly but helpful trick is to deliberately think what the command is doing in English when you use the command.

Eventually, you can just think in English while you work and habit gets your fingers to do the right thing.[2]

[2] This is also (and especially) true for keyboard shortcuts, which are an important tool if you use a terminal-based text editor.

Basic syntax

Syntax: `[command] [option(s)] [argument(s)]`

Example: `find output -name 'robustnesscheck87.png'`

Sometimes commands have implied arguments when unspecified:

Example: `cd` (implied $\sim$)

Getting help

Ask for help on a command with `-h` or `-help`.

Example: `git commit -h`

When `-h` doesn't work, there often is a manual entry that can be accessed with `man [command]`.

Example: `man cat`

Bonus (on macOS or Linux): install cheat.

Moving around

`pwd`: "Where am I?"

`cd`: "Move to"

`ls`: "List"

`.` : "Here"

`..` : "Backwards one directory"

$\sim$: "User directory"

\: "Base directory"

Example: `cd ../..` "Move backwards two directories"

Example: `ls input` "List files in the input folder"

Moving around

Warning: \ tells *bash* to interpret the next character 'literally'. This is required with spaces.

If you named a folder `A Folder`, you would need to `cd A\ Folder`. [3]

[3] Another benefit of using the command line is that it encourages you to name your folder `my_folder` instead.

Moving around

`find`: "Find (files or folders)"

Example: `find a_folder -iname '*.txt'`

"Find text files in `a_folder`"

Example: `find . -type d -iname 'a_folder'`

"Find folders with name `a_folder`"

Example: `find . -size +100k`

"Find files that are larger than 100kb"

Creating and destroying things

`mkdir`: "Make a directory"

`touch`: "Create a file"

`rm`: "Destroy a file"

`rmdir`: "Destroy a directory" (fails if not empty)

`rm -r`: "Destroy this and everything it contains"

Example: `touch hello.txt` "Create an empty .txt file called `hello.txt`."

Warning: `rm` really does destroy, it doesn't move to Trash

Other common operations

`cp`: "Copy this"

`cp -r`: "Copy this and everything it contains"

`mv`: "Move (or rename, if moving to same directory)"

Example: `mv hello.txt goodbye.txt` "rename `hello.txt` to `goodbye.txt`"

Wildcards

If you've stopped paying attention and don't know about these, pay attention again.

- `*`: "I can be anything"
- `?`: "I can be one thing"

Example: `mv input/*.txt .` "move all text files in the input directory here"

Example: `ls hello_?.txt` "lists `hello_1.txt` but not `hello_10.txt`"

Echo and cat

`echo` "Print"

`cat` "Print the contents of a file"

`>` "write to file (overwrites if it exists)"

`>>` "append to file"

Example: `echo 'Hello' > hello.txt`

"Create text file with 'Hello' on the first line"

Example: `cat hello.txt > goodbye.txt`

"Copies everything from `hello.txt` to `goodbye.txt`"

Example: `echo 'Goodbye' >> goodbye.txt`

"Adds 'Goodbye' to the last line of `goodbye.txt`"

Head and tail

`head` "Show me the beginning of the file"

`tail`[4] "Show me the end of the file"

Example: `head 'goodbye.txt'`

"Show me the first 10 line of `goodbye.txt` "

Example: `tail -n 3 'hello.txt'`

"Show me the last 3 lines of `hello.txt` "

[4] *Note:* `tail` is useful for debugging. We often do `tail stata_do_file_that_failed.log` to quickly see the final error message.

Sed, grep, and piping

`sed` [5] and `grep` are great and can do almost any task in their domain (though you may need to Google around some to find the right flag).

`sed`: "Substitute"

`grep`: "Find (text)"

`|`: "Pass to the next command"

Example: `grep -Rl ‘‘hello’’`

Example: `sed -i ‘‘s/hello/goodbye/’’ hello.txt`

Example: `grep -o ‘‘input/[A-Za-z0-9]*.[a-z]’’ | xargs sed -i ‘‘s/input/output/g’’`

[5] I've seen people on Stack Overflow advocate for `awk` over `sed` on a case-by-case basis (and it may be especially better for `.csv` editing), but I have less experience with `awk`.

Aliases

Aliases are self-defined shortcuts.

You define these in a script file that executes every time you launch your terminal.

The file is in your user directory and is called `.bashrc` (or `.zshrc`).

Syntax: `alias="[full command]"`

This is also where your `PATH` lives (I can add notes on this).

GitHub, the right way

Configuration

The first time you use GitHub on the command line, you need to configure your username and email.

```
git config --global user.name ‘‘your_username’’
```

```
git config --global user.name youremail@example.com
```

GitHub uses personal access tokens (PATs) instead of passwords.

Follow the steps outlined here to get a PAT to use when prompted for your password.

Cloning a repository

Go to the repository, click on the big green ‘Code‘ button, and copy the link.

Open the command line, navigate to the parent folder that you the project to exist in, and type:

```
git clone [link]
```

It should download the project into a folder with the repository name.

Commits and branches

When in a GitHub repository, you live at a certain `commit` on a certain branch.

Commits are nodes that correspond a set of changes to code.

If you only commit to the `main` (default) branch, your project history is just a series of changes from the first commit to your most recent commit.

Commits and branches

You can create also a new `branch`, which will become its own series new commits that are not incorporated into the main code.

When you decide to incorporate the new branch into the `main` (on GitHub this normally means that a teammate has reviewed and approved the changes via a pull request), you can `merge` the changes from the branch into `main`.

The basic toolkit

`git pull`: "Grab and apply the changes that I don't have"

`git branch [branch name]`[6]: "Create a branch"

`git branch`: "Show me branches that I could switch to"

`git switch [branch name]`: "Switch to a certain branch"

`git status`: "Show me the state of directory"

`git restore [files]`: "Undo changes that I have made"

`git clean`: "Undo changes that I have made"

`git add [files]`: "Stage these changes to be committed"

`git commit -m ''commit message''`: "Commit staged changes"

`git push`: "Send commits that I have made off to GitHub"

[6] Branching off of the commit that you are currently located at.

Git checkout and other commands to check out

`git checkout [branch name]`:
"Switch to the branch"

`git checkout [branch name] -- [file]`:
"Restore the file to the state it is at in the specified branch"

If you are spending a lot of time integrating changes from one branch into another, look into `rebase` and `cherry-pick`.

Worktrees

I just discovered the `git worktree` command, and it is great.

If you work on multiple branches simultaneously often and don't want to constantly be switching between them, start using worktrees.

git diff

git bisect

git add –patch

GitHub CLI

Also look into GitHub CLI.

It allows you to do most of what you can do on the website from the terminal.

Example: `gh repo create new_repo`

Example: `gh issue list -a ljmotley`

Example: `gh browse`

Automate what can be automated

Automation

When you are writing code for a project, ideally everything that goes into the final product should be able to be run with one process.

There are a variety of ways to implement this; the most basic is a `run_all` script in your language of choice.

But there are more versatile, language-agnostic tools such as Makefiles and Taskfiles if you want to explore these options.

Living in the terminal

Discovering your inner hacker

- Working from one location
- Notice what you do slowly
- Command-line text editors
- Terminal multiplexers
- Customizing your environment

Thank you