Session 02: Your Development Environment

wifi: GA-Guest, yellowpencil

GENERAL ASSEMBLY

Today's session plan

1800-1820	Standup & Review
1820-1845	Introduction to the CLI
1845-1915	CLI practise
1915-1940	Break
1940-2000	Introducing Git & GitHub
2000-2030	Git setup & practise
2030-2100	Using Git for DS37
Homework: The command line murders	

At the end of the session, you will be able to ...

Use Bash to perform file operations

Understand why Git and GitHub are powerful version control tools

Use Git locally for version control

Pull from GitHub repositories

Data Science Part Time

Review

Which of these best describes a supervised learning task?

A. Using a target to predict an independent variable

B. Using features to predict a response

C. Finding patterns and groupings in datasets

D. Automating tasks performed by humans

Which of these best describes a supervised learning task?

A. Using a target to predict an independent variable

B. Using features to predict a response

C. Finding patterns and groupings in datasets

D. Automating tasks performed by humans

Who can give me an example of a categorical variable?

Which of these is a regression task?

A. Predicting whether someone will default on a loan or not

B. Predicting a house's sale price

C. Clustering customers into similar groups

Which of these is a regression task?

A. Predicting whether someone will default on a loan or not

B. Predicting a house's sale price

C. Clustering customers into similar groups

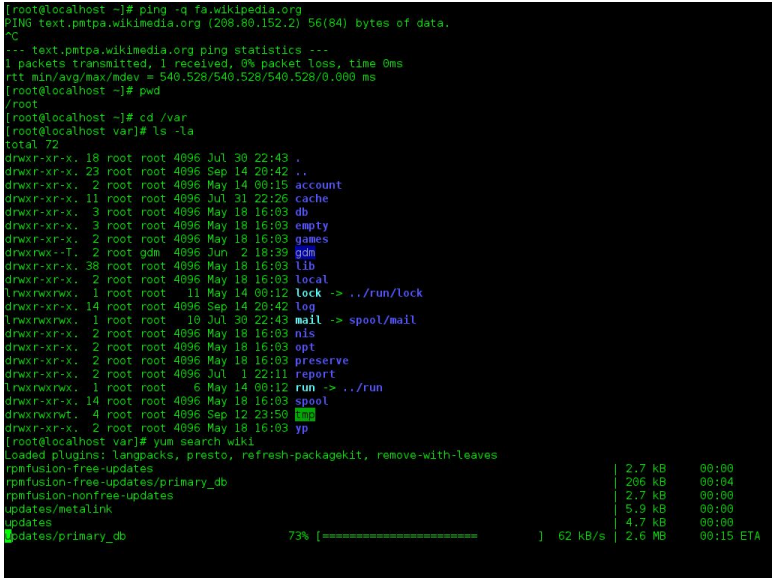
bit.ly/ds37-session-02

Data Science Part Time

Introduction to the CLI

What is the Command Line Interface?

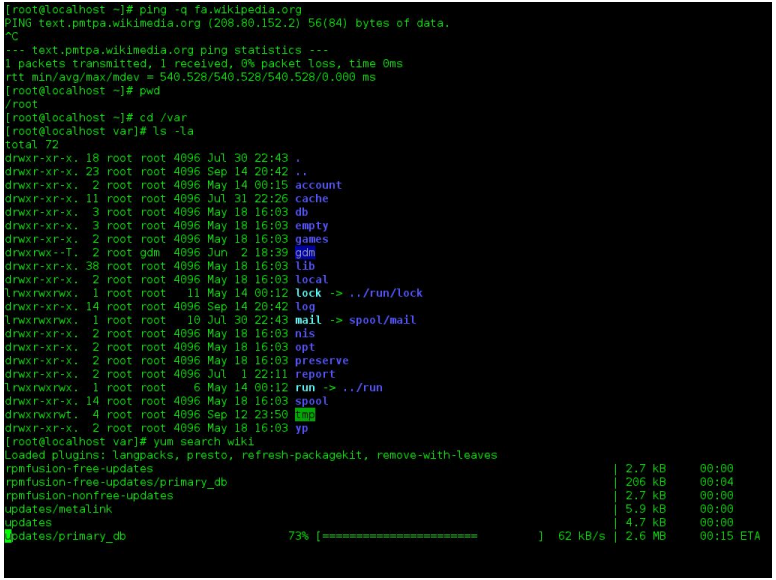
There was a time when computers didn't have graphical interfaces; everything had to be done through the **command line interface.** This is the closest you can get to having a 'direct line' to your computer's processor.



What is the Command Line Interface?

Nowadays, we can use graphical user interfaces (GUIs or ‘gooeys’) to perform most tasks on our computers.

But there are some tasks that can be performed faster and more efficiently through the CLI.



Partner Exercise:

Using a GUI

In pairs, perform the following tasks on one of your laptops. Count the number of mouse-clicks needed to perform each task:

1. Creating a new folder in your Documents folder and naming it ‘cli’
2. Navigating to the new ‘cli’ folder
3. Creating a new file called ‘test.txt’ in the ‘cli’ folder
4. Adding the line of text ‘hello world’ to your new ‘test.txt’ file
5. Downloading the file at this URL to the ‘cli’ folder, and giving it the filename ‘parliament28012020.txt’: https://hansard.parliament.uk/debates/GetDebateAsText/8b238af9-c5e4-461d-8e77-ba8e587776d4
6. Previewing the first few lines of this file

Why is the CLI useful?

Using the command line interface, we can perform **all** of the tasks we carried out in the previous exercise, but faster and with fewer clicks.

The CLI can be used to:

Create, move and copy files and folders
Download files from the web
Keep track of different versions of files
Run Python scripts
Search files

...And much more

Launching the command line interface

The CLI (often also called the **shell** or **terminal**) can be accessed in different ways depending on your operating system.

On a Mac:

Open the ‘terminal’ application.

On Windows:

Open the ‘Git Bash’ application you downloaded before this course.

The language of the CLI

Different programming languages can be used in the command line interface.

We'll be using **bash**, which is one of the more widely used languages.

Computers Out:
Launching the CLI

Let's check we're all able to launch the terminal on our laptops.

For Mac Users: Launch the 'terminal' app on your laptop.

For Windows Users: Launch the 'git bash' app on your laptop.

Now let's try out our very first command in bash!

```
echo hello world
```


Filepaths

A filepath is the position of a file or folder in a system.

Folders are often referred to as **directories**.

Files and directories can be read, written, moved, or deleted by referencing their **path**.

Filepaths

Absolute paths are given relative to your computer's **root** directory; the starting point from which all other folders are defined. This is usually shown as having the filepath '**/**'

```
/Users/Lucy/desktop/a/b/c/file.txt
```

Relative paths are given relative to your current **position** in your file system.

```
c/file.txt
```

Note, your **root** directory is not the same as your **home** directory, which is usually found at /Users/[Your Username] or **~** for short.

Computers Out:

File operations in bash

Try out the following commands along with your instructor:

```
pwd
```

```
cd
```

Independently, find out what the following commands do:

```
cd .
cd ..
cd ~
cd /
```


Terminal shortcuts

Here are some useful shortcuts that make working with files more efficient in the terminal:

`↑` Retrieves the last command you ran in the terminal

`.` Shorthand for your current directory

`..` Shorthand for the directory that your current directory is in

`~` Shorthand for your home directory

`/` Shorthand for your root directory

`ctrl+C` Gets you out of trouble

Computers Out:
Exercises

With the command line, use `cd` and `pwd` to navigate to these directories and confirm you've reached them:

Your home directory
A folder of your choice that's inside your home directory, using a relative path
Back to your home directory, using `..`
The directory that your Jupyter notebook from Tuesday's class is in

What does the following command do?

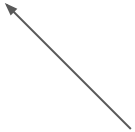
```
open .
```


The structure of a bash command

```
command -option input
```

The **command** name depends on the operation we're performing, e.g. echo, pwd, cd

Options come after the command name, and alter the behaviour of the command



This is the **input** we're giving the command, e.g. the filepath to change directory to or the message to echo

Computers Out:
Command structure

1. Use the `ls` command with no inputs or modifiers, to **list** the contents of your **current working directory**

2. Now re-run the `ls` command. This time, add an input (sometimes called an **argument**) after the command name, to list the contents of your Documents folder like this:
   `ls -a ~/Documents`

3. What does the `-a` modifier do when added to the `ls` command like this:
   `ls -a`

4. What does the `-l` modifier do when added to the `ls` command like this:
   `ls -l`

5. What does the result of this command tell us about using multiple modifiers together?
   `ls -al`

Guided Walk-Through:
Making and deleting files

It's easy to create blank files using bash, and even to start adding content to them. Open up your file browser next to your Terminal window.

Watch your instructor and then follow along on your own laptops:

`touch myfile.txt` creates a file

`echo hello world >> myfile.txt` appends a line of text to the file

`cat myfile.txt` displays the contents of the file

Guided Walk-Through:
Making and deleting directories

It's easy to create directories using bash. Watch your instructor and then follow along on your own laptops:

`mkdir mydirectory` creates a directory

`rm mydirectory` deletes a directory, but only if it's empty

`rm -rf mydirectory` deletes a directory even if it contains files/directories

Be very, very careful when using rm

Guided Walk-Through:
Moving and copying

Watch your instructor and then follow along on your own laptops:

`mv source destination` moves a file or directory from one location to another (this can also be used to rename files, if the source directory is the same as the destination directory)

`cp source destination` copies a file or directory from one location to another

Computers Out:
Practising bash

Try to complete the following exercises, using only bash

Base:

1. Create a directory in your ‘Documents’ folder called ‘datascience’
2. Create a file called hello.txt in that directory
3. Write the following lines of text to hello.txt in three stages:
   ```
   Hello, its me
   Ive been wondering if after all these years youd like to meet
   To go over everything
   ```
4. View the contents of the file to confirm the lines have been added correctly
5. Rename hello.txt to adele.txt
6. Copy adele.txt to your Documents folder

Stretch:

1. Change directories to your Documents folder. What does the command `ls *.txt` do? How about `ls a*` ?
2. Change directories to a folder with plenty of files in it. Using the results of the previous question, can you list:
   - All the .doc files in this folder
   - All the Excel files

Guided Walk-Through:
Downloading from the web

Watch your instructor and follow along:

1. Make sure you're in your newly created 'datascience' directory

2. Let's try out the curl command. Run `curl example.com`
   What does this command do?

3. Now let's add an option. What does `curl -o example.html example.com` do?

4. Let's use `cat` to inspect the contents of example.html

5. What does `head -n 5 example.html` do? When might we use it instead of `cat?`

Computers Out:
Practising bash

Try to complete the following exercises, using only bash

Base:

1. Make sure you're in your newly created 'datascience' directory
2. Remember the file you manually downloaded and renamed from the web earlier? Perform the same task, but using `curl` in a single step: download the file at https://hansard.parliament.uk/debates/GetDebateAsText/30d231a9-deb5-4d1c-9dae-3aa2bd2b4bff and call it 'parliament28012020.txt'

Stretch:

Can you use the `wc` command to find out the following from your Hansard file:

1. The total number of words
2. The total number of lines

Discussion: CLI vs GUI

As a class, let's make sure we understand the bash commands we've just learned. How could I perform the same function as each of these bash commands, using only the GUI and not the terminal?

- `cd`
- `pwd`
- `mv`
- `cp`
- `curl`
- `cat`

Intro to Python

Version control with git

Discussion:

Why do we need version control?

Imagine these scenarios:

1. You've written a report at work. You're making some some tweaks to it, but your want to save previous versions in case you need to revert back to an older version that your boss prefers.

2. You're building an Excel spreadsheet that contains lots of formulas. You're worried that you'll make a mistake when writing a formula that will break the whole spreadsheet, so you want to save previous versions of the spreadsheet as a safety ned that you can revert back to if needed.

3. You're playing a video game. If you 'die' in the game without saving your progress, all your progress since the last save will be lost.

When was the last time you encountered a scenario similar to the ones above? How did you manage different versions of your work to get around these problems?

Version control done badly

Most of us will have done something like this at some point:

```
Report_v1.doc

Report_v1_MA_edits.doc

Report_v3_final.docx

Report_v3final with charts.doc

Report final.doc

Report FINAL FINAL.doc
```


Version control

Version control is the process of managing subsequent **versions** of files on your computer.

This can be done **manually** (a bad idea) or with the help of **version control software**.

One of the most widely used pieces of **version control software** is called git.

We'll be using git throughout this course.

Introducing git

Git is the most widely used version control software.

We interact with git through the **terminal**.

Git allows us to...

- Track changes to folders and files on our laptops
- Create ‘save points’ that we can roll back to

In this course, you’ll be using git to...

- Obtain the resources (slides, Jupyter notebooks, data) for every session
- Track changes to the folder where you keep your General Assembly resources, so you don’t have to worry about ‘breaking’ anything irreversibly

Group Exercise:

The git workflow

Can you re-order these sentences into a workflow that makes sense?

Save a copy of the current state of your entire directory, that you can roll back to later if needed.

Tell git which files in that directory you want it to track.

Become worried you'll irreversibly 'break' your code.

Tell git to start 'watching' a directory on your laptop.

Realise you need to use git for version control.

Start working on a new data science project, involving lots of Python scripts.

The git workflow

Start working on a new data science project, involving lots of Python scripts.
Become worried you'll irreversibly 'break' your code.
Realise you need to use git for version control.

Initialising

Tell git to start 'watching' a directory on your laptop.

Staging

Tell git which files in that directory you want it to track.

Committing

Save a copy or 'snapshot' of the current state of your entire directory, that you can roll back to later if needed.

Initialising a repository

When you decide you want to start using git for version control on a particular project, all the files for that project will typically be in one folder (if you're organised).

We need to explicitly tell git that we want it to start watching or tracking that folder.

This is called **initialising a git repository (or 'repo')**.

A **git repository** is just a folder on your laptop that we've told git to start watching.

We do this using the `git init` command.

Guided Walk-Through:
Initialising a git repository

Follow along with your instructor:

1. Create a directory called **git_practise** in your **Documents** folder
2. Change directories to git_practise
3. Create three files called **file1.txt**, **file2.txt** and **file3.txt** in that directory
4. Check the contents of the directory using `ls -a`
5. Now, tell git to start watching this directory by typing `git init`
6. Check the contents of the directory again using `ls -a`
7. What's changed?
8. Run `git status` to check in on the current state of your git repository. What do you think the message means?

Staging changes

It's not enough to tell git to start watching a directory by running `git init`.

We also need to tell git **which** files in that directory we want it to track.

This is called **staging** files.

Only files that have been **staged** can be included in our 'snapshots'. So if we want to be able to roll back to previous versions of a file, we need to make sure it's been **staged**.

We do this using the `git add` command.

Guided Walk-Through:
Staging files

Follow along with your instructor:

1. In your **git_practise** directory, run `git add file1.txt`
2. Now run `git add file2.txt` and `git add file3.txt` (for bonus points, what would be a faster way of doing this using the * operator?)
3. Run `git status` to check in on the current state of your git repository. What's changed since last time we ran this command?
4. Let's add some lines of text to our three files.

Commits

Once we've made some changes to our work, it's likely we'll want to take a 'snapshot' or save a copy of all the files in our directory that we've **staged**.

Taking successive snapshots of a directory means that:

- We have a record of the changes we've made to the files in our directory over time
- We can roll back to previous snapshots if we mess up our work, accidentally delete something or irreversibly 'break' something in our files.

A snapshot is called a **commit**. We do this using the `git commit` command.

Think of making a **commit** as like saving your progress in a game. You can always roll back to previous saves (or commits) if you die (or mess up your work) further on in the game.

Guided Walk-Through:
Making commits

Follow along with your instructor:

1. In your **git_practise** directory, run `git commit -m "Added some text to all three files"`
2. Run `git status` to check in on the current state of your git repository. What's changed since last time we ran this command?

Making further commits

Once you've initialised a repository and staged some files to be tracked, it's up to you to:

(a) Commit your changes as often as you think is sensible. Make sure your commit messages are descriptive enough to explain what's changed in a specific commit, just like good code comments.

(b) Stage any new files you create, that you would like to start tracking.

The majority of the time, the only two commands you'll use in git are git add and git commit.

If all your work goes perfectly smoothly and you don't make any mistakes*, you'll never need to 'roll' back your repository to a previous commit.

*This never happens.

Guided Walk-Through:
Making further commits

Follow along with your instructor:

1. In your **git_practise** directory, add some more lines of text to each of your three **.txt** files.
2. Commit your changes by running `git commit -m "Added even more text to all three files"`
3. Why is git complaining at us?
4. Stage all the text files in our repository either individually with:
   ```
   git add file1.txt
   git add file2.txt
   git add file3.txt
   ```
   or with a single command `git add --all`
5. Now when we run `git commit -m "Added even more text to all three files"` the commit goes through

Shortcuts

It’s inconvenient to have to **stage** files before making each **commit**.

We can combine the process of staging and committing using the option `git commit -am`

The **‘-a’** option stages any files that were previously staged, and then commits those files or takes a snapshot of them.

Guided Walk-Through:
Rolling back to previous commits

Follow along with your instructor:

1. In your text editor of choice, delete the last line of text in **file1.txt**.
2. Imagine you go for your lunch break and you later realise that deleting this line of text was a terrible mistake.
3. Run git log and find the SHA 1 associated with the **snapshot or commit that we want to roll back to**.
4. Now run `git checkout <your SHA 1 here>`
5. The last line of text in **file1.txt** has reappeared, because we're now viewing a **previous snapshot of our repository**.
6. To return back to the current state of our repository, run `git checkout master`

The git workflow

Three important steps in the git workflow are:

1. **Initialising**
   `git init`
   Tell git to start ‘watching’ a directory on your laptop.

2. **Staging**
   `git add`
   Tell git which files in that directory you want it to track.

3. **Committing**
   `git commit`
   Save a copy or ‘snapshot’ of the current state of your entire directory, that you can roll back to later if needed.

Guided Walk-Through:
Diffing

It’s possible to see what’s changed between successive versions of the same file using git.

1. Let’s use git log to find the hash numbers of two successive commits

2. We can then use `git diff <hash1> <hash2>` to see what’s changed between the two commits

Computers Out:
Git in practise

Perform the following tasks using bash and git commands only*:

1. Create a new directory inside your Documents directory called ‘ebooks’ and change to that directory
2. Download the file at this URL as ebook.txt: http://www.gutenberg.org/cache/epub/1497/pg1497.txt
3. Use `head` to inspect the file and figure out what’s in it
4. Use `wc` to figure out how many lines and words the file has
5. Initialise your directory as a git repository
6. Add ‘ebook.txt’ to the staging area and make a commit, with a sensible commit message
7. Use `open ebook.txt` to open up the file in your default text editor
8. Give your laptop to the person sitting next to you. Ask your partner to **secretly** change one word of their choice somewhere in the file, and replace it with another or phrase of their choice. Then ask your partner to save and close the file.
9. Make another commit
10. Use `git log` and `git diff` to find out exactly what your partner changed in your file

Collaboration with GitHub

Sometimes, keeping track of different versions of our work **locally** on our laptops isn't sufficient.

We might want to collaborate with colleagues, or we might want an extra safety net in case we lose or break out laptops.

This is where GitHub comes in.

GitHub is not the same thing as git!

Git is a piece of version control software that runs **locally** on your computer.

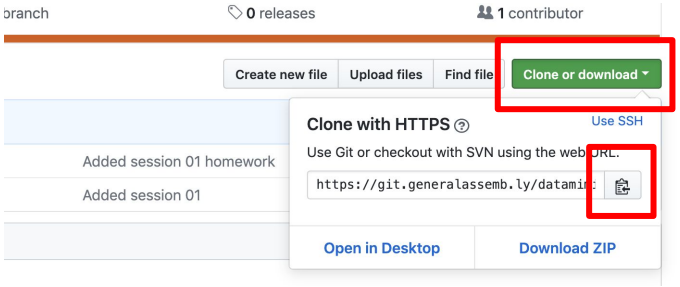
GitHub is a website where people can **upload** or **push** their git repositories.

If a repository is **public**, other people can download or **clone** it. Every time the repository on GitHub changes, anyone who has **cloned** it onto their own laptops can **pull** the latest changes from the online repository.

Guided Walk-Through:
Setting up GitHub

In this course, you'll be **pulling** lesson resources (slides, Jupyter notebooks, data) every session from our course GitHub repository. Follow along with your instructor to get set up.

1. Visit https://git.generalassemb.ly/datamimi/ga-ldn-ds37
2. Select 'clone or download' and hit the 'copy' button
3. In your Terminal, navigate to your Documents folder
4. Write `git clone` and then paste in the copied text from GitHub
5. Use `ls` to figure out what's in the cloned repository. Compare it to what's on GitHub.
6. Use `git log` to see the entire commit history for this repository
7. Watch as your instructor **pushes** the resources for session 2 to GitHub.
8. Run `git pull` to **pull** these changes.
9. Use `ls` to confirm that you've successfully pulled the new material



Using GitHub in this course

At the start of every session, change to your **ga-ldn-ds37** directory and run `git pull` to download the newest material for the session from GitHub.

As you make your own changes to the resources in the folder (e.g. by completing the exercises in Jupyter Notebook) remember to make regular commits to save your progress. One commit at the end of every session is a bare minimum.

You will need to **commit** your changes in your local copy of ga-ldn-ds37 **before** each time you pull from GitHub.

Intro to Python

Let's Review

At the end of the session, you will be able to ...

Use Bash to perform file operations

Understand why Git and GitHub are powerful version control tools

Use Git locally for version control

Pull from GitHub repositories

Feedback

Take a minute to fill out our end of week survey:

http://bit.ly/ds37weekly

Coming up next week...

- Python foundations
- Getting data using APIs
- Final project ideas

Homework

Try out this fun game, the Command Line Murders, to hone your git and bash skills:

https://github.com/veltman/clmystery


GA