LECTURE NOTES: Version Control with Git

Instructor: *Leila Inigo de la Cruz*

Last update: *25-10-2023*

These lecture notes for the beginner-level of one of the lessons of the Software Carpentry. These notes assume the use of Git terminal for Windows. The context and flow of this lesson have been addapted to better fit the audience.

PREPARATION

To set up the command history on two terminals do the following:

1. On main terminal:

```
$ export PROMPT_COMMAND="history -a; $PROMPT_COMMAND"
```

2. On second (history) terminal:

```
$ tail -f ~/.bash_history
```

Windows Terminal (Preview) [Keyboard shortcuts]

Useful shortcuts for the App Windows Terminal (Preview) on Windows 10.

Action	Shortcut
Split pane horizontally	Alt + Shift + `-`
Split pane vertically	Alt + Shift + `+`
Close a pane	Ctrl+Shift + `w`
Move pane focus	Alt + `Arrow keys`
Resize the focused pane	Alt+Shift + `Arrow keys`

PART 1

INTRODUCTION [10 min]

Version control systems start with a base version of a document and then record changes you make each step of the way.

Presentation notes

1. SETTING UP GIT (Lecture begins) [6 min]

Sometimes, the shortcut on the Windows menu for Git Bash won't work. In such a case: go to the installation folder (usually `C:/Git`) and double-click on `git-bash.exe`

Key points: Use `git config` with the `--global` option to configure a user name, email address, etc.

Git Command Syntax `git <command> [options]`

Explian syntax using "how to get help" as example.

- How to Get Help with the Commands
  - show help for all commands: `git --help`
  - show help for specific command: e.g., `git init -h`

a. Setting up a Username and Email

- For the ones that already have done that , you can always check your global configuration by typing:

```
git config --list
```

```
git config --global user.name "github-name"
git config --global user.email "github-email@mail.com"
```

b. Configure Line Breaks

Use Notepad++ to demo the line breaks

Explanation: Every time you press return on your keyboard you insert an invisible character called a line ending. Different operating systems handle line endings differently.

When you're collaborating on projects with Git and GitHub, Git might produce unexpected results if, for example, you're working on a Windows machine, and your collaborator has made a change in macOS.

You can configure Git to handle line endings automatically so you can collaborate effectively with people who use different operating systems.

- For Windows `shell` `git config --global core.autocrlf true`
- For MacOS and Linux `shell` `git config --global core.autocrlf input`

c. Check Global Settings

```
git config --global --list
```

2. CREATING A REPOSITORY [4 min]

Key Points: `git init` initialises a repository. Git stores all of its repository data in a hidden `.git` directory.

Use case: Two researchers/programmers are developing code to analyse the inflammation data used in the Python session.

a. Make a Directory on the Desktop

```
mkdir ~/Desktop/patients
```

b. Initialize the Repository

```
cd ./patients/
git init
```

Repository, or Repo, is another word for project

c. Check Content and Status

```
ls -a
git status
```

Explanation: initialising (creating .git files) for every folder inside a repo is redundant and bad practice. Git uses this special subdirectory to store all the information about the project, including all files and sub-directories located within the project's directory. If we ever delete the .git subdirectory, we will lose the project's history.

3. START TRACKING CHANGES [10 min]

Key Points: How git track changes and the modify-add-commit cycle.

a. Create a Python Script to Count Lines

The script will count lines from the standard input. For this Python must be accessible from the terminal.

- Create and modify file

```
nano calculate_mean.py
```

- Type code

```python
import numpy as np

def calculate_mean(data):
    mean = np.mean(data)
    return mean

def main():
    # Sample dataset as a numpy array
    data = np.array([10, 20, 30, 40, 50])

    # Calculate the mean
    mean = calculate_mean(data)
    print(f"Mean of the dataset: {mean}")

if __name__ == "__main__":
    main()
```

- Check that the file is in the repository `shell      ls` #### b. Test the Script

```
# check Python is accessible from Git bash
 python --version
```

```
python calculate_mean.py
```

or

```
python3 calculate_mean.py
```

- possible error: numpy is not installed, then we should type in the bash

```
pip install numpy
```

*introduce another change to check that output a correct mean value with a known dataset , data=np.array([10,20])

c. Check Git Status

```
git status
```

d. Start Tacking a File Using git add-commit

```
git add calculate_mean.py
```

> Talk about the files that git can track :

```
- text files (.txt, .md , .html)
- source code (.py , .R, etc )
```

> For example .png , .avi , .exe can not be tracked by git "in depth" , git only will notice we have modified them if needed.
>
> However there are some tools developed by git now that can also track .ipynb (jupyter notebooks) which is a binary file.
>
> Ignore the warning about replacing LF with CRLF. | **Warning: LF will be replaced by CRLF in calculate_mean.py. The file will have its original line endings in your working directory** |

e. Commit Changes Creates a snapshot of the changes in the repository's history three.

```
git commit -m "create  script calculate_mean.py"
```

`git commit -a` or `--all` "stage all changes and write them to history"

A good commit message is short (< 50 characters), and completes the sentence: 'This will..' **message** [use slide]

Explanation of **staging**. The working directory, the staging area, and the git history. `git add` is used to define which files we want to commit. `git add` specifies what changes to stage; `git commit` takes a snapshot of the changes and writes them to the repository's history. [Use illustration]

Explanation of **modify-add-commit** cycle. [Use illustration] , go to a slide and show what adds and commit means graphically

Questions?

4. MAKING OTHER CHANGES [5 min]

We know that a good coding practice is using comments to describe our code. Let's add some helpful comments to our script, e.g., author, python version and a short description of what the script does.

- Add a docstring bellow the calculate_mean function:

```
""" This function computes the mean of a one dimensional array.
Input: An one dimensional array
Output: The mean of the array
"""
```

- check status `shell` `git status` > Explanation: Notice that modified files are automatically tracked by Git, however they are not automatically committed. This is desirable because is up to the user to decide when and what to commit to the repository's history.

b. Check Differences (review changes)

```
git diff
```

Shows the difference between the current state and the most recently saved version (last commit)

c. Add and Commit

```
git add calculate_mean.py
```

d. Diff vs Diff –staged

```
git diff
```

- this does not show the difference with staged change

```
git diff --staged
```

- shows the difference with staged change

  `git diff`: shows the difference between no-staged changes and the previous commit. `git diff --staged`: shows the difference between staged-changes and the previous commit.

Finally, commit the changes:

```
git commit -m "add docstring".
```

5. GIT & EMPTY DIRECTORIES [4 mins]

a. Create a Directory 'treatments'.

Ask students to create a new directory, try to stage it, and check the status.

Solutions:

```
mkdir treatments
```

- check the status and add the directory. No changes are added

  ```
  $ git status
  $ git add treatments
  $ git status
  ```

  Explanation: git doesn't track empty directories. Git tracks the content of a file and its name (including its path).

b. Create Files on the Directory and Add Changes

```
touch treatments/aspirin.txt treatments/advil.txt
```

c. Stage All Files in the Directory

```
git add treatments/
git status
```

d. Commit

```
git commit -m "add some treatments for patients"
```

display a tree of the directory to see the behavious so far :

```
sudo apt-get install tree
tree .
```

or in git bash

```
cmd //c tree //a
```

> Go again to the illustration to explain ### 6. IGNORING THINGS [6 mins]

Say you have files you don't want to tack with git.

> We'll create some fictitious data files for now.

```
mkdir data
touch data/a.dat data/b.dat big-data.zip
```

> **Important:** git is not good for tracking large dataset, especially binary files. This is because binary files will be fully copied to the repository history when committed, and changes to tracked binary files will cause Git to save a copy file of the file for every commit. Therefore, increasing the size of the repository rapidly.

b. Create .gitignore File

> At the root of the repository, create a `.gitignore` file, and type in it the path and name of all files and directories you don't want to tack.

```
nano .gitignore
```

Type:

```
big-data.zip
data/
```

A good practice is to use comments to explain why files and directories are ignored. For example:

```
# data files:
big-data.zip
data/
```

- Check git status

```
git status
```

- Add and commit .gitignore*

```
git add .gitignore
git commit -m "Ignoring data files"
```

c. Check What's Being Ignored

```
git status --ignored
```

> come back to the ilustration

> **Questions?**

FIRST BREAK

PART 2

7. EXPLORING THE HISTORY [16 min]

a. Checking the Log

```
git log
git log --graph [optional]
git log --oneline
```

it is like the CHANGELOG.txt file

```
git log > changelog.txt
```

Explanation of output: unique ID and list of commit messages.

Explanation content of Directory and where changes are stored.

Paging the log. **Q**= quit, **spacebar**= next page, /=search word, **N**=navigate thru matches. `git log --oneline`, limit output to one line. `git log --graph` print a text graph of the history tree.

b. HEAD

In the following parts (b-e) is more important to **put attention** than to follow along. Put attention, follow along only if you won't lose focus.

The **HEAD** is a **pointer** that refers to the *current active branch* in the git, which can be that last commit we made or the last commit that was checkout into the working directory. We haven't created more branches and the current history tree only contains one single branch, called by default **master** or **main**. In our case HEAD points to the most recent commit in the *master/main* branch. We can refer to the most recent commit using HEAD as an identifier.

c. Add more to the Documentation of calculate_mean.py

```
nano calculate_mean.py
```

```
# Usage:
# data=np.array([0,10,20,30])
# mean=calculate_mean(data)
```

d. Check Differences Compared to HEAD

```
git diff HEAD calculate_mean.py
```

This is the same as not using HEAD, because the HEAD is currently pointing to the latest commit. However, we can use **HEAD** to check the difference between the current state of `calculate_mean.py` (in the working directory) and previous commits.

```
git diff HEAD~1 calculate_mean.py
git diff HEAD~3 calculate_mean.py
```

`HEAD~1` compares with the last commit. `HEAD~3` compares to 3 commits ago.

e. Comparison Using the commit IDs

"You will have to reference the Hash explicitly if you want to see the `diff` of a file that was not changed between the last commit and the one before it (HEAD~1)."

Usage:

git diff `<start-commit-SHA>` **HEAD** `<file>`

```
git log --oneline # [copy an ID to compare]
```

```
git diff commit-id calculate_mean.py # [use ID for first commit]
git diff commit-id HEAD calculate_mean.py # [use ID for first commit]
```

Wrap up this section by showing an illustration of the git history tree

7. REVERTING CHANGES [9 min]

Follow along

BEFORE THAT: Add and commit the changes calculate_mean.py.

```
git calculate_mean.py
git commit -m "add usage"
```

a. Revert to Older Versions Using an Identifier.

One way to revert changes is using the commit ID. Restore latest version in the history tree using the `checkout` command.

```
git log --oneline # [copy ID of "description input"]
```

```
git checkout <id--commit> calculate_mean.py # [will revert changes, use firts commit ID]
cat calculate_mean.py
```

b. Restore the version without any Docstring [Optional]

Additional example, used only if on schedule. Ask participants to do them by themselves.

```
git checkout commit-id calculate_mean.py
```

Changes go to the staging area; they are not committed.
However, we always can go back to any version we have committed.
To go back to the newest version, check out to HEAD.

- check out to HEAD

```
git checkout HEAD calculate_mean.py
cat calculate_mean.py [notice the file is back to the newest committed version]
```

- Add and Commit the newest version

```
git add calculate_mean.py
git commit -m "update author's name"
```

EXERCISE: Create Repository and Track Changes [15 mins]

a. Explain exercise in plenary Exercise description (slides):
https://docs.google.com/presentation/d/17vM2uc_wvCcw7mVMqsNud71K_QZTlcXM4rD2DygkAtk/edit?usp=

b. Helpers and partcipants go to a Breakout session

Suggestion: Share your screeen, and ask participats to try things first by themselves, then show them how to do it. Give them about 1 minute per activity `[1-6]` and then show them the answers one at the time.

c. Answers

- **Create new repository, use the modify-add-commit cycle, and recover older versions.**

  1. Create and initialize a repository called 'my-repo'.

     ```
     mkdr my-repo
     cd my-repo/
     git init
     ```

  2. Create a files `research.txt` with the sentence: **Science is awesome**

     ```
     nano research.txt
     ```

     Inside the file type the following and save changes: `shell Science is awesome`

  3. Add and commit the changes. Remember to use a meaning message.

     ```
     git add research.txt
     git commit -m "add awesome science"
     ```

4. Change sentence in ‘research.txt’ to: **Science is messy** `shell nano research.txt` Change text to this and save changes: `shell Science is messy`
5. Add and commit. `shell git add research.txt git commit -m "change to messy science"`
6. Revert changes to the very first version of ‘research.txt’, and commit. `shell git log --oneline # find and copy ID of the firts commit git checkout <commit-ID> research.txt # revert changes cat reseach.txt # check version has been recovered git commit -m "recover awesome science" # commit recovered version`

- **Check your history log – you should have 3 commits**

```
git log # print full log
git log --graph # print log as text-graph
git log --oneline # print short version of log
```

PART 3

8. REMOTES IN GITHUB [20 min]

Students use their GitHub account to create an empty repository. They follow instructions to push their local copy to the remote.

Explain what GitHub is [**slides, 2 min**]

a. Create GitHub Repo

Go to Github and create an empty and public repository called `patients-analysis`.

- Repo description: *analysis of treatments for inflammation*

b. Add Remote to Local Repo

b0. Conect to GitHub via SSH [Technical Break, 30 min]

Recently, GitHub requires authentification via SSH to do pulls an pushes, but not for cloning. **Use illustrations** to explain what a SSH connection entitles.

To connect via SSH do the following:

- Create a Key-pair inside the `.ssh` in the Home directory

```
# move to Home directory
cd ~
# create key
ssh-keygen -t ed25519 -C "your_email@example.com"
# save to the default location and file name: ~/.ssh/id_ed25519 , or put a specific add
```

- Check the keys have been created

  ```
  ls ~/.ssh/
  ```

- Start the `ssh-agent` and add private key to agent:

  ```
  # start agent
  eval "$(ssh-agent -s)"

  # add private key
  ssh-add ~/.ssh/id_ed25519
  ```

  > Instruct SSH to use key files in different locations: `ssh -i <path/private/keyfile>`

  Info on how to (start the ssh-agent automatically)[https://docs.github.com/en/authentication/connecting-to-github-with-ssh/working-with-ssh-key-passphrases#auto-launching-ssh-agent-on-git-for-windows] Mac and Linux user don't have to worry about this.

- Copy public key to GitHub:

  ```
  clip < .ssh/id_ed25519.pub
  ```

  > if clip does not work then use

  ```
  cat  .ssh/id_ed25519.pub # copy its content to github
  ```

- Go to GitHub, explain the basics of the interface and add the SSH key.

Profile > Settings > SSH and GPG keys > New SSH key > Add SSH key

- Test SSH connection

  ```
  ssh -T git@github.com
  ```

  More information on working with (SSH keys and GitHub.)[https://docs.github.com/en/authentication/con
  to-github-with-ssh]

  Check the info on (Troubleshooting SSH[https://docs.github.com/en/authentication/troubleshooting-ssh]) for GitHub.

b1 Add the remote Move back to the repo directory: '~/Desktop/ > In your local repository (on the terminal), add the remote repository and push the content.

- Connect to remote `shell     git remote add origin git@github.com:<user-name>/<repo-name>.g`

- Check that remote was added `shell git remote -v git branch -M main # [will change the name of the main branch of the repo to make it more friendly] git push -u origin main`

c. Check the Content Repositoy is in GitHub

- Go back to your repo page and refresh the browser.
- To pull changes from the remote: `shell git pull origin main`

  Questions?

d. Exploring the GitHub GUI (optional)

SECOND BREAK
> Invite several participants as collaborators to the repository `workshop-check-in`

PART 4

9. CONFLICTS (Demo) [15 min]

Explanation of when a conflict can happen: "A conflict arises when two collaborators make changes to the same line in a file, or when a file has been deleted by one collaborator, but edited by another."

Demo using the `calculate_mean.py`. A helper and the Instructor will create a conflict and present a solution.

a. Create conflict

-
-
-
-

b. Solve conflict

-

10. COLLABORATING [15 min]

Explain the concept of social coding. [1.5 min]

a. Clone workshop-check-in Repository

Move to the Desktop and clone the workshop-check-in repo. Share the link of the repo in the chat -> `https://github.com/leilaicruz/workshop-checkin`

```
cd ~/Desktop
git clone git@github.com:leilaicruz/workshop-checkin.git
```

b. Create a Check-in file

Make a copy of `check-in/template.md` in the same Directory; remane the file using a unique name (e.g. three first letters of your name and the last two digits of your phone number. Mind the file extension ".md")

```
cd workshop-check-in
cp check-in/template.md check-in/<my-nickname-file>.md
```

c. Edit your Check-in file

Edit `<my-name-file>.md` and change the content. You'll see some hints

```
nano check-in/<my-name-file>.md
```

d. Pull, Add, Commit, and Push to the remote A basic collaborative workflow using git is:

- "Update the local repo with git `pull origin main`,"
- "Make changes and stage them with `git add`,"
- "Commit changes with `git commit -m`, and"
- "Upload the changes to GitHub with `git push origin main`"

Example:

```
git pull origin main
git add .
git commit -m "check manuel in"
git push origin main #[This works only if participants are added to the repository as collab
```

Ask a participant to push their changes to remote, and show the changes int the GitHuh GUI.

e. [Optional] Demo Create branches and Pull requests

Create a branch from what was pull from main

```
git pull origin main
git branch leila71
git checkout leila71
```

c. Edit your Check-in file

Edit `<my-name-file>.md` and change the content. You'll see some hints

```
nano check-in/<my-name-file>.md
```

d. Pull, Add, Commit, and Push to the remote A basic collaborative workflow using git is:

- "Make changes and stage them with `git add`,"
- "Commit changes with `git commit -m`, and"
- "Upload the changes to GitHub with `git push origin leila71`"

Demo pull request on GitHub

11. LESSON SUMMARY [2 min]

- Repository initialization `git init`
- Git records changes via commits to the history tree
- Remember the **modify-add-commit** cycle and the **working directory-stage-commit** areas
- Don't include large datasets in your repositories. Set a `.gitignore` file
- Remotes store copies of the git repositories (e.g., GitHub, GitLab)
- Collaborative workflow using branches: **pull, add, commit, push, pull request**
- Be aware of *conflicts*

12. [Optinal] Licencing and Citation [5 min]

Explaing the importance of licencing and citation for Open Science. Share template for compliance with TU Deflt software policy: https://github.com/manuGil/fair-code

13. Q&A

Participants ask questions.