Introduction to version control (Git)

Learning outcomes

- Navigate through folders using basic command line commands
- Create a new folder and file using your command line
- Use basic git commands in your command line to commit code
- Start using best practices as you develop your projects

Recap from last session

- A file is composed of 3 main components. What are they?
- What is a function? Why do you need it?
- What are some best practices of writing a function?

Working with the command line

This chapter is arguably the most likely one to make you feel like a hacker. The command line is the black window that works as a textbased application for viewing and manipulating files on your computer, and so much more. Other names for the command line include cmd, CLI, prompt, console or terminal. For the purpose of this exercise, you can choose to use either the integrated terminal in VSCode, or open a new standalone window. Please follow the appropriate instructions below:

VSCode

Go to View -> Terminal.

Windows

Try Start menu -> All Programs -> Accessories -> Command Prompt. Alternatively, go to the Start menu or screen, and enter "Command Prompt" in the search field.

Mac

Go to Applications -> Utilities -> Terminal. Alternatively, enter "Terminal" into Spotlight Search (Cmd + Space).

Linux

Try Applications -> Accessories -> Terminal, or Applications -> System -> Terminal.

Navigating through folders using the terminal

Now that you've opened a terminal window, let's see where you are. Type in the following command and see what that prints:

For Windows users: `cd`

You should see the same filepath as before the > prompt printed on a separate line. That is your "current directory".

For Mac/Linux:

```
pwd
```

`pwd` stands for "print work directory", therefore the output should be the path where you currently are in the filesystem, otherwise known as "current working directory". Start exploring where you are, by typing:

Windows: `dir`

Mac/Linux: `ls`

This will list the directories and files in your current working directory.

To navigate through folders, you can use the command `cd` ("change directory"). You can also navigate to a specific folder (`cd path\to\here` on Windows or `cd path/to/here` otherwise), or go back up one level (to the parent directory) using `cd ..` . Have a play around with these commands to get yourself familiar with navigating your filesystem. See examples below.

Windows:

```
cd Desktop dir
cd ..\..
cd
```

Mac/Linux:

```
cd Desktop ls
cd ../.. pwd
```

> *Bonus*: If you work on a Mac/Linux machine, try:
>
> ```
> cd pwd
> ```
>
> What does that print?
>
> On Unix systems, `cd` on its own takes you to your home directory.

Creating/deleting files and folders using the terminal

Now let's create a new folder. Navigate to a location of your choice using `cd` and then run:

```
mkdir new_dir cd
new_dir
```

You'll notice that spaces are not tolerated in the names. You can enforce them with "\", but for the sake of simplicity, you could use underscores instead.

Now let's create a new empty file, `hello.txt` , in your new directory.

Windows: `type nul > hello.txt`

This writes the contents of `nul` to `hello.txt` , i.e. an empty file.

Mac/Linux: `touch`

```
hello.txt
```

If you're interested in learning more about `touch` , feel free to have a look here.

You can quickly inspect the contents of the file by pasting it in your terminal window like so:

Windows: `type`

`hello.txt` Mac/Linux:

```
cat hello.txt
```

Let's add some text in your file and double-check the result: echo "Hello"

```
>> hello.txt
```

echo is a command that prints out the input string it is given, while >> append to the end of the file. If you want to overwrite the file completely, you'd use simply > . You can double-check the result by using type or cat as highlighted in the step above.

Finally, let's clean up after ourselves and remove the files and folders using:

Windows:

```
del hello.txt cd ..
rmdir new_dir
```
Mac/Linux:

```
rm hello.txt cd ..
rmdir new_dir
```

Note that rmdir only works for removing empty directories. If you wish to delete a folder along with all its contents, you should use /s on Windows, or -r flag ("recursive") on Mac/Linux:

```
rmdir /s new_dir rm -r
new_dir
```

Introduction to pip

You will be relieved to hear that you won't always have to code everything from scratch, but instead re-use functionality that other people have written. These are typically referred to as *packages* or *libraries*, and can be downloaded and installed in your local environment. pip enables the installation of such third-party packages.

Let's go ahead and install some of the packages you will need later in the course: pandas and matplotlib .

First, make sure you're in the correct conda environment. Check your command line to see whether (coding-session) appears at the beginning of the line, which is the name of the environment you created in the pre-work. If that's not the case, you can switch to the right environment by running the following in your command line:

```
conda activate coding-session
```

Make sure pip corresponds to Python 3 and not Python 2 by running

```
pip --version
```

This will show pip 19.3.1 from /path-to-site-packages/pip (python 3.6) (pip version might be different, but that's okay).

Now we can go ahead and install the desired libraries:

```
pip install pandas pip install
matplotlib
```

You will notice that you are now allowed to do

```
import pandas import matplotlib
```
in your

python code, with no errors.

Congratulations, you are now a command line wizard!

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. An example of a tool that has version control is Google Docs. When working with Google Docs, version control can keep track of the different versions of draft for you and allows you to revert changes you made. As you can imagine, such a tool is useful when writing code as well.

Git is a widely used version control system (VCS) and you can think of it as Google Docs for code. Git enables you to revert changes you made to your code back to a previous state, or to compare changes you or someone else has made over time. This is especially useful when you encounter a bug, because it allows you to find what piece of code was changed and when.

These are several reasons for why we need a version control system:

- Backup and restore
  - Files are saved as they are edited, and you can jump to any moment in time. Need that file as it was on Feb 23, 2007? No problem.
- Synchronization
  - Git lets people share files and stay up-to-date with the latest version.
- Undo
  - When you have messed up with a file, you can easily discard your changes and go back to the "last known working" version.
- Track changes.
  - As files are updated, you can leave messages explaining why the change was made (it is stored in the VCS log, not in the actual file). This makes it easy to see how a file evolves over time, and why.
- Track ownership
  - A VCS tags every change with the name of the person who made it.
- Branching and merging
  - You can branch a copy of your code into a separate area and modify it in isolation (tracking changes separately). Later, you can merge your work back into the common area.

Installing and Initialising Git

Let's first install Git - there are several ways to do it. Git may already be installed, so first check if it is by running the following command in the

command line: `git --version`

If it returns a Git version, you are good to go. Otherwise, follow the instructions in the pre-work to install it.

In order to start using Git, you first need to create a new Git repository. In this session we will be using the command line to create files and folders. First create a folder called `git_practice` and go inside the newly created folder.

```
mkdir git_practice cd
git_practice
```

So far this `git_practice` directory is just a regular folder, not a Git repository. To initialise git, run the following: `git init`

If you run the command below you can see a hidden folder `.git` , which indicates that Git is initialised. The `.git` folder contains all the necessary information for your project in version control, such as commits, remote repository address (don't worry if you don't know them) etc. It also contains all the git history that you can roll back.

For Windows users: `dir`

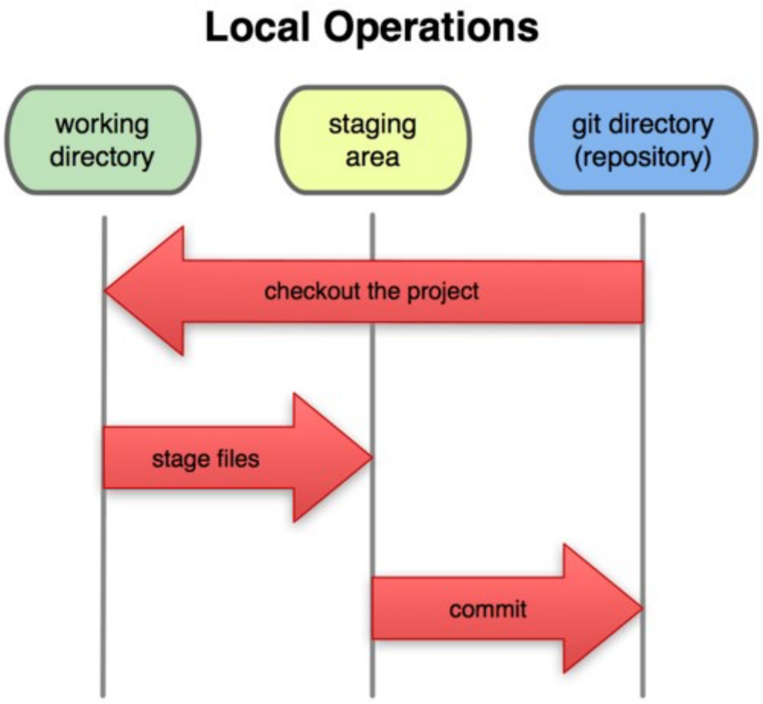
```
/adh
```

For Mac/Linux users: `ls -a`

Git commit

At a high-level, Git can be thought of as a timeline management utility. Git deals with data as a series of snapshots that form a timeline. These snapshots are called *commits* and they are the core building block units of a Git project timeline. A commit is created with the `git commit` command to capture the state of a project at that point in time.

The workflow of Git commit is illustrated below.



> Image source: https://git-scm.com/book/en/v1/Getting-Started-Git-Basics

Git has three main states that your files can reside in: modified, staged and committed.

- Modified means your code/files have changed, but not saved by Git yet.
- Staged means the changed files have been marked and will be committed to the next commit snapshot.
- Committed means the code/files have been saved by Git.

Just like it's good to test your work in smaller bits as you go along, it's best practice to commit frequently, when you've got something smaller working. This way, if something goes wrong, you don't have to start at the beginning because you forgot to commit your code.

Let's work through an example first and come back to the details later. First, let's add a text file with some text.

For Windows users:

```
type nul > hello.txt echo "Hello" >>
```

hello.txt For Mac/Linux users:

```
touch hello.txt echo "Hello" >>
```

hello.txt If you now run the

following: `git status`

you will see that `hello.txt` is an untracked file. This means that git will not keep track of changes made to this file. To make sure git will start tracking this file from now on, run:

```
git add hello.txt
```

which will add your files to the staging area. If you now run: `git status` again, you will see

that hello.txt is in the changes to be committed stage.

Finally, let's commit the change to Git

```
git commit -m "my first commit"
```

where -m stands for "message", followed by the actual message. Make sure to add a meaningful commit message, which will help your team members and future-self to understand what kind of change you are making in the commit.

Having committed, you can see the commit history by running

`git log` which will show the history of commit messages and IDs. You can go back to any point or snapshot with that ID.

Task

1. Add more changes to the same text file and commit to git.
2. Add another change to the text file, but before committing run:

```
git diff
```

this will show the changes Git will commit.

Setting up a new repository on Github

A git repository is the thing that holds all of your project files and folders. You'll need to set one up for your course project. While you can do this on your computer and link it with your GitHub account, today we'll create the repository in GitHub and "clone" it to your computer instead.

> Note: Git vs Github: A common confusion for beginners is the difference between Git and Github. Whilst Git is a version control system (tool), Github is a hosting service for Git repositories (web service). Github is also a good place for developers to showcase your work to potential employers and friends.

1. Open your browser and log into your account on GitHub (http://github.com)
2. Click on the tab called "Repositories" 3. Click on the button that says "New"

4. Give your new repository a name, and then click "Create Repository" (the repository will be created as public by default).

Don't worry about the other fields for now.

GitHub now shows us a page with a bunch of different options for uploading our code. We don't want to upload any code yet; we just want to make sure we've got a repository for our project on our computer that's linked with the one we just made on GitHub.

1. Open up your command line
2. Navigate to your Python course folder using cd
3. Create a copy of your new GitHub repository on your laptop by running the following command

```
git clone https://github.com/YOUR_REPOSITORY_NAME
```

This creates a new folder for your project, which contains your git repository. It's also where you'll put all your project files. You can check that the folder was created by using ls in your command line.

> Note: if you ever wonder if the directory is a git repository or not, type

```
git status
```

> on the command line in the directory. If this is a repository you will see the status of it otherwise git will show a message saying that location has not been initialized as a repository.

Task:

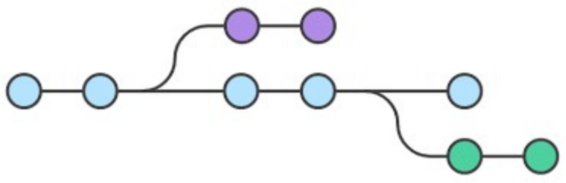
1. Find an interesting repository on Github.
2. Click "Clone or download" in the repository, and copy the URL.
3. In your command line run the following: `git clone https://github.com/USER_NAME/REPOSITORY_NAME` and you should get the source code of that

project on your computer.

Advanced Git and Github (optional)

Branches

Branching is a very useful feature in version control systems. It allows you to branch out and diverge from the main line of development so you can continue developing without affecting the main (or master) branch. This also enables different team members to work on building different parts of a project at the same time. With branches, you can test and commit changes on a separate branch of code before merging those changes into your master code branch.



Let's list all Git branches currently available in our project. Run the following command: `git branch`

which will show a branch called `master` .

Let's create a new branch with Git

`git checkout -b new-branch` which does two

things:

1. Create a new branch from the `master` branch (`-b` stands for "branch")
2. Switch from `master` branch to `new-branch` branch.

To verify this, run `git branch` again, you should see the newly created branch. Remember, if you're using branches, never delete your master branch!

Suppose you have made some code changes in the `new-branch` branch that you now want to `merge` back into your `master` branch. You can do this as follows:

First switch back to the master branch: `git checkout master` Then run the following: `git merge new-branch` which tells Git to merge the `new-branch` branch into the `master`

branch.

Pushing commits to a remote branch

Next, let's push commits made on your local computer to a remote repository in Github. The command we use is `git push` , which takes two arguments:

- A remote name (e.g. origin), usually the location of your Github project A branch
- name (e.g master)

Using the above example, you run: git push origin master which

pushes your local changes to your online repository.

Task:

Let's use the cloned git project in the previous step.

1. Add a new commit in your branch (master branch).
2. Run git push origin master , and see if you can see the update in your Github project page.

Managing conflicts in git

It is very likely that if you work on a coding project in a team, different team members will make different changes to the same file. Sometimes these changes will conflict with each other, and someone will get a message in their command line saying they can't push their code because it is not up to date. This is called a *merge conflict*.

Merge conflict means they can't merge their changes with the master branch. When this happens, Git will also add some markers in your file to tell you where the conflicts are. The person who tries to sync/merge last will be the one that has to deal with the merge conflict.

Merge conflicts are annoying, but you can fix them. Here's how:

1. Open the problem file in your text editor
2. Merge conflicts have lines around them that look like this <<<<<<<<<< . Your document may have a few lines like that, depending on where the conflicts are. You might see ======= as well.
3. Choose the lines of code you want to keep, editing them if you need to.
4. Once you've got the code the way you want it, remove the <<<<<<<<<<< bits you see and save your file
5. Use git add <flename> to add the updated file to the staging area
6. Commit and push the change

Avoiding conflicts in git

You should avoid merge conflicts by making sure you're using the most up-to-date code from GitHub and regularly pushing code/submitting pull requests.

Remember that committing code just commits it to your local branch. Your team members won't see anything until you push it to GitHub. A *pull request (aka. PR)* is a request you make when you're proposing your changes to your team's code, and requesting that someone pulls in your contribution to GitHub.

1. Get the latest code from your team's repository on GitHub by using git pull origin master to update your local repository 2. Make the changes you want to make to your code
3. Add any changes to the staging area, then commit them to your computer's repository
4. Push your changes to your team's repository on GitHub
5. git push - sync upload the changes back to Github so others can see them

Homework and additional resources

Git has a lot more features, and see the following resources if you would like to learn more:

- Udacity Version Control with Git free online course
- Codecademy Learn Git online course
- Git cheatsheet
- Learn Git Branching: an interactive tutorial and a great way to get more familiar with branching.

Reference

The content of this course is developed based on the following references:

- A Visual Guide to Version Control
- Udacity Version Control with Git
- Git SCM documentation
- Github documentation