**Week 8: Lab pre-read**

In this session we will:

* Learn about GitHub as a version control system
* Create your Github accounts and practice Git instructions
* Collaborate with your peers on a small project

## 1. Creating a GitHub account and working with Git

Now that you have had a go at basic programming, let’s get you set up with a GitHub account. This will be useful for you as you continue your journey as a programmer. GitHub allows you to collaborate with others as well as keep all your code stored in an organised way with the history of changes all in one location.

Open <https://github.com/join> and add your details to create an account. Once your account is created, install [git](https://git-scm.com/book/en/v2/Getting-Started-Installing-Git) on your computer. Git is an open source version control tool. A convenient way to give Git instructions is through the terminal which are then interpreted and executed by the Git shell that you have just installed. Thus, it is a good idea to familiarise yourself with some [basic commands](https://ubuntu.com/tutorials/command-line-for-beginners#1-overview) used in the terminal.

#### 1.1 Configuring your identity in git (first time usage only)

The first things that needs to be configured after installation are the identifiers that git will be using to track the user making any commits going forward. These only need to be set once on a system and are stored by git. You can set your username and email address with the following commands:

git config --global user.name "Your Name"

git config --global user.email "you@example.com"

#### 1.2 Create a local git repository

Open a terminal (**Ctrl/Command+Alt+T**) and navigate to where you want to create the project with the *cd* (change directory) command followed by the path, for instance if you want to create the project folder at the Desktop run the following command:

cd ~/Desktop

Next, create a new folder using the *mkdir* (make directory) command and navigate to it using the *cd* command:

mkdir folder\_name

cd folder\_name

Next, we will initialise the root of the folder as a git repository (i.e. a repository managed by git version control system) using the *git init* command:

git init

#### 1.3 Add a file to the git repository

To add a file to the folder, you can copy-paste the hello.py file you created earlier or create a new one. Note that git will not know which files exist unless we ask it to *track* the file. You can use the command *git status* to see which files git knows of. You will see that that git prompts you that it knows you created a new file but the file is not yet *tracked*.

To begin tracking a file, one must *add* or *stage* the file using the following command:

git add <filename>

You can now use the *git status* command to check that the status of the file has changed. Adding a file to git saves the current snapshot of the file into the commit memory of git. In other words, it notifies git of the files or changes in files that have been put on stage for committing.

To add multiple files you can use the following command

git add .

#### 1.4 Commit a file to git

In order to register the changes that have been put onto the staging area, one must use the following command:

git commit -m "Your message about the commit"

This instruction does two things, (a) it links the content/changes in the staging area with an explanatory text provided by the user within quotation marks, and (b) it commits the changes to the git memory such that the current state of local repository, from the point of view of git, includes the stages changes.

Note that the message provided by the user should reflect an explanation of what one is committing to the repository. This helps collaborators in understanding what has been contributed in clear words.

#### 1.5 Creating a remote repository on GitHub

What you have created until now is a *local* repository which is git initialised and has its current state registered with git. It is now time to share your code with the rest of the world. For this, the local repository needs to be linked with a *remote* repository that is hosted on GitHub.

Naturally, the first step would be to create this remote repository on GitHub. To do so,

1. Login to your GitHub account and select **New Repository** under the **+** sign in the top right corner.
2. Add a name and description for the repository.
3. Finally press **Create Repository**.

#### 1.6 Linking the local and remote repositories

Two repositories are now in existence, one locally in which rests your code, and one remotely housed on GitHub that has the README.md. **The next and important step is to synchronise the two such that the remote repository serves as a public image of the local repository.**

git remote add origin <link\_to\_github\_repo>

*Remote* - the remote flag allows you to create a link to a remote repository (such as the one you have just created on GitHub) using a placeholder name as a bookmark to refer to the URL of that repository. (You can check all remote connections using the git remote -v command.)

*Origin* - refers to the central repository. This keyword is used when a repository is cloned to point back to the cloned repository, allowing a clear reference to pull remote changes from and push local changes to the central repository.

When initialising a connection with a remote or central repository, it is good practice to first update or pull any changes made upstream (in the remote or central repository) in order to have your local repository in sync with the central repository. This is especially important when working with collaborators, as others may have been pushing changes onto the central repository.

git pull -u origin main

**\*\*Note**, if you have created a repository on GitHub with a README.md file while setting up and have already made commits on your local repository before connecting to the remote, you may face some merge conflicts while pulling. This happens because you have two separate version histories (one local and one remote) that have no idea of the other’s existence and how they are related. Since the version histories do not match, a merge conflict is created. Two ways of tackling this are:

1. If you have already made commits on the local repository, do not create a remote repository with a README.md file while setting it up and rather push it with the rest of the local repository to an empty remote repository.
2. If you have already set up a remote repository with a README.md file, initialise an empty local repository (i.e. without any commit history) and pull from the remote repository before adding any new commits in the local repository.

A safer option is git fetch which allows you to only download the updated repositories but does not automatically merge them with your local repository, in case you want to first review the changes before manually merging them with your local repository (refer to git merge). Read more about how git pull and git fetch work differently [here](https://stackoverflow.com/questions/292357/what-is-the-difference-between-git-pull-and-git-fetch) and how you can see the differences after fetching [here](https://stackoverflow.com/questions/1800783/how-to-compare-a-local-git-branch-with-its-remote-branch).

After syncing your local repository with the remote repository, you can now push any changes you have made to the repository upstream to the central repository for collaborators to access.

It is also good practice to have a **README.md** file with a description of your project included in your local repository before pushing.

git push -u origin main

Here the *-u* flag is shorthand for *--set-upstream* and adds a tracking reference for the upstream branch that you are pushing to.

Now you can see your repository uploaded on GitHub.

#### 1.7 Branches in git

When you are working on a project with several collaborators, it is normal to have many people working on a specific task and making additions and changes to the code at the same time. This is where branches come in to make tracking these changes easy.

When you create a new repository a **main** branch gets created by default. This is where the most *stable* or final version of your code is stored. Branches are where different *states* of your codes are stored. This way git can easily keep track of when, what and by whom any new changes are committed to the project and from which commit your branch 'branched' off of, so it knows the history behind all the files. And when you have gone over the new commits and want to add them to the main branch you can choose to **merge** them.

Let’s create a new branch with some minor changes to your hello.py program. Open the file again and add another print statement of your choice below the existing one and then exit. Now create a new branch by running the following command:

git checkout -b <my branch name>

This will create (-b) and switch you (checkout) to the new branch that you created. You can check which branch you're on by running the git branch command. To push this change to GitHub run the following command:

git push -u origin <my branch name>

Once the changes have been pushed you will see a prompt in GitHub that a new branch has been added. Now you can compare the changes and make a **pull request** by clicking on the green button in the prompt.

#### 1.8 Pull Requests

A pull request (or PR) is a way to alert a repo's owners that you want to make some changes to their code. It allows them to review the code and make sure it looks good before putting your changes on the primary branch. Here you can describe the changes or additions you have made for the owner (or yourself) to review and keep a track of. Once you create the pull request, you will see an option to **merge pull request** which will allow you to merge the changes into the main branch. Once you confirm the merge, you can delete the branch to simplify your repository.

#### 

### Advanced Git instructions

#### 1.9 Reset

This command allows you to discard changes that have been made to the working (local) directory but are not yet committed. Three modes of operation stand out, (a) git reset --soft, (b) git reset --mixed and (c) git reset --hard.

The first instruction performs a soft reset whereby the working directly is left untouched and only the local HEAD is updated to the last commit. In other words, the git reset --soft instruction works as the *opposite of git commit*. It *uncommits* the changes made to the files, while pointing the local HEAD to the last commit but keeps the files staged.

The second instruction performs a mixed reset whereby the working directly is left untouched and only the local HEAD is updated to the last commit. In other words, the git reset --mixed instruction works as the *opposite of git commit* as well as *git add*. It *uncommits* and *unstages* the changes made to the files, while pointing the local HEAD to the last commit, regarding all changes on top as unrecorded/new.

The third instruction performs a hard reset and must be used with extreme caution. The entire state of the working directory is changed to match the local HEAD which is moved to the last commit. In other words, all changes – staged or unstaged – are dropped, all untracked files are deleted, and the state of the working directory is as it was after the last commit.

#### 1.10 Revert

This git instruction is to reverse a commit done in the past. The revert command is doing the opposite of a commit in practice. It may be thought of as an undo operation but unlike an undo operation the previous commit is not removed from the version history, rather it removes all changes made in the specific commit and adds a new commit with the changes removed. This way the version history is consistent in remembering what exactly was added or removed at any point.

The revert command requires a commit reference to be added to refer to the commit history (this can be accessed by using the git log command). The reference given should specify the changes that you want reversed. Note the previous commit in the current branch can also be reverted by using HEAD as a reference for where the head of the branch currently lies.

git revert <reference>

[Note: When reversing a merge commit, it is not obvious which changes need to be undone i.e. those on the main-line or on the sub-line. For example, let’s call the initial state of code as C\_0. Let’s say the code has changed along the main-line through commits c\_m1 and c\_m2 into C\_m. The same code C\_0 has changed along the sub-line through commits c\_s1, c\_s2 and c\_s3 into state C\_s. We then merge C\_m and C\_s and commit the merge operation. When reverting this specific merge commit, it is not obvious whether one should reverse changes c\_s1, c\_s2, and c\_s3 made w.r.t to the main-line or changes c\_m1 and c\_m2 considering the sub-line as the reference. To assist with this, we have the ‘-m’ flag with the git revert command. The flag ‘-m’ must be used with a numerical value where a value of 1 sets the mainline as the reference point, and 2 sets the sub-line as the reference.]

#### 1.11 Cherry-pick

This git instruction is to selectively apply the changes that were part of a previous commit at a certain state of the repository.

## 2. Play area

You can use this simulation cum visualisation [tool](https://git-school.github.io/visualizing-git/#free) to practice and get used to how git manages the version history with the different commands. [Cheat sheet](https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet) for reference.

## 3. Exercise

Write a code that performs one of the following exercises and push the local repository you have created to GitHub. Collaborate with a classmate, pull their repository to your local machine, make some additions to the project and push it back to the central repository. You can add a collaborator by going into the settings of your repository and under the Collaborators tab adding people.

1. Anagram: Write a code that accepts two phrases and prints if they are anagrams or not
2. Palindrome: Write a code that accepts a phrase and checks if its a palindrome or not

## 4. Peer Assessment

Grading to be done for your assigned working partner based on the implementation of the exercises and of the git/GitHub collaboration process [here](https://forms.office.com/r/vhDeLc1cpR).