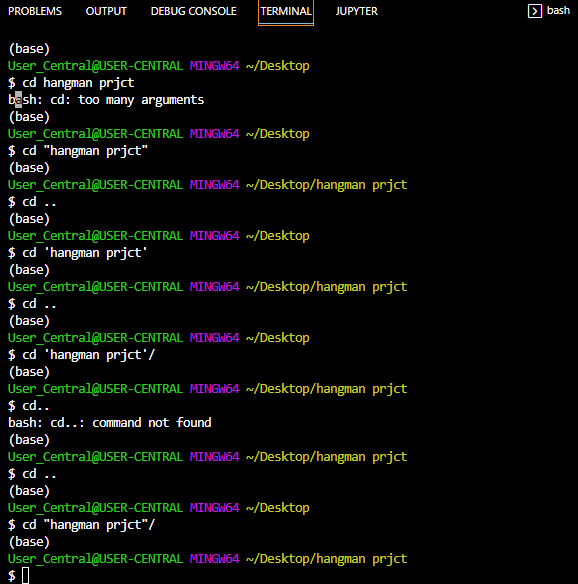
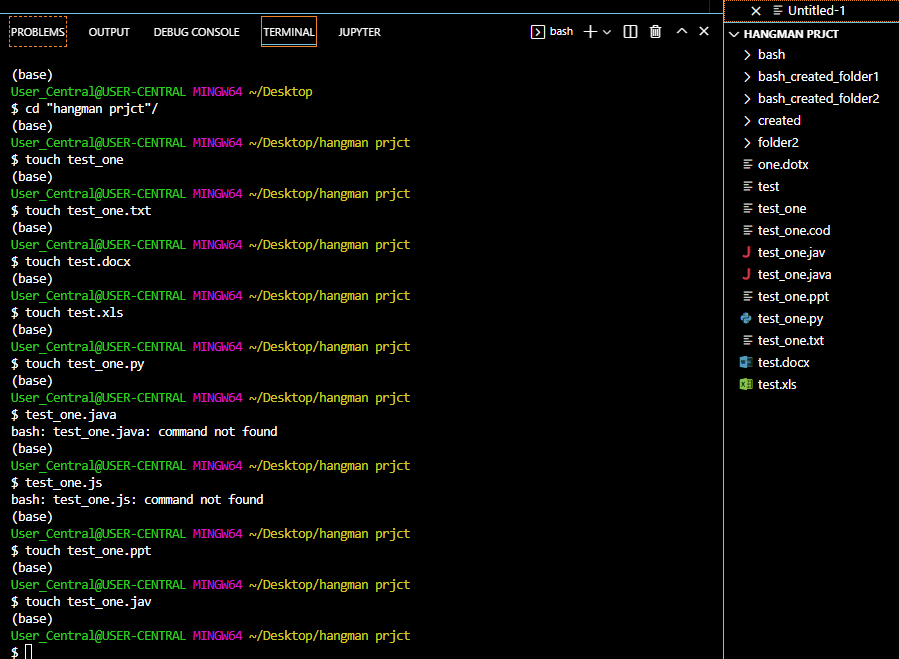
*The* **cd** *command*



Wrong way to cd (change directory ) to a folder that has two names not joined with an underscore i.e folder name is not one word.

Correct way to label and cd to a folder i.e in quotes like a string; along as the folder name is not one word



Git

git is currently the most utilised distributed **version-control system** (VCS) and the de facto standard for code collaboration globally.

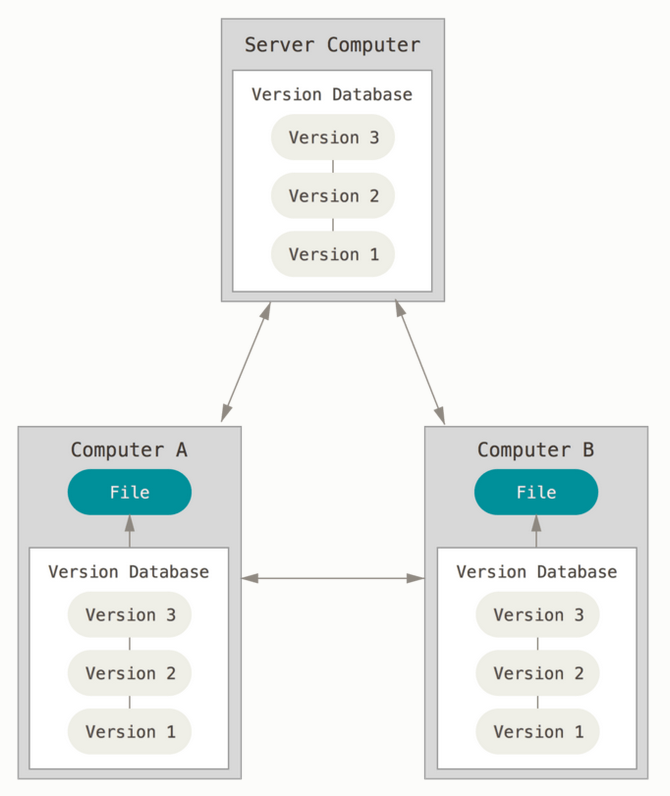
As a program, git can be run in the command line, which we are slightly familiar with. If you are on a Windows machine, use gitbash; otherwise, use the default terminal.

Version-Control System

First, what is a VCS?

As the name suggests, a VCS refers to a software utilised for tracking changes to code and maintaining the records of saved versions.

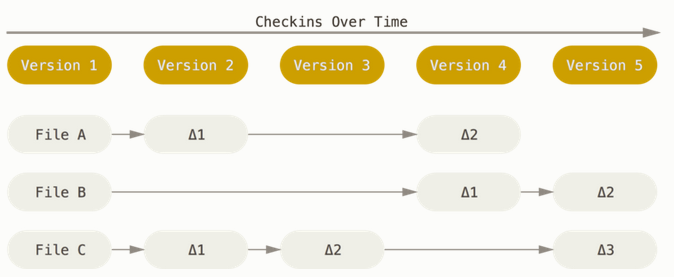
A **VCS** records changes to file(s) over time, thereby providing constant access to previous code **versions.**



Git is considered to be **distributed** because each node (client) **mirrors the full repository and its history.**

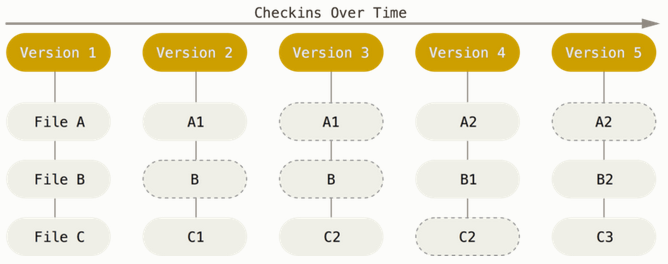
Git Features

Other than git, other VCSs, which are comparatively unpopular currently, **store information about file changes** (see below).



Thus, these VCSs only consider a file if a change has been made. In other words, the new state does not consider files that have not changed.

Conversely, git records 'snapshots' of the whole code, known as commits. Think of a commit as a picture of all the files at a certain point. With that picture, git can always revert to that state.



Repositories

When you tell git to start keeping track of the files in a folder (or directory), you move the commits to a git **repository**, in which git stores the snapshots of the files in the working directory.

As mentioned previously, git is a distributed VCS; however, that does not mean that you change the state of the central server whenever you work on your repository. When you work on a repository, your changes are saved locally and will not be reflected on the central server until you push them.

Note that git only adds data. This implies that the operation of removing a file could be considered as 'add file deletion'. Thus, you will not lose data if you commit your changes frequently. You **ESPECIALLY** will not lose your data if you push your changes to a central server (as you probably can tell, we are gradually approaching GitHub).

State of any File in git

The files in a repository can be in one of three stages:

* Modified: You changed the file but are yet to commit the changes to your database.
* Staged: You have marked a modified file in its current version as being ready to go into your next commit (snapshot).
* Committed: The data are safely stored in your local database.

For clarity, here is an example of the conventional flow of a file in a repository:

1. The file is created or modified. Thereafter, git compares the changes in the current directory with those in the last snapshot and notices some changes.
   * The revised file is now labelled as **modified** because it changed with respect to its last snapshot.
2. When you have completed a session of revisions on the file, you can mark the file as being ready for the snapshot, thereby positioning it in the **staged** state.
3. git takes the snapshot of the file and stores it in the repository. The file is now **committed.**
   * The next time you change a file in the working directory, it will enter the modified state, and the process repeats once more.

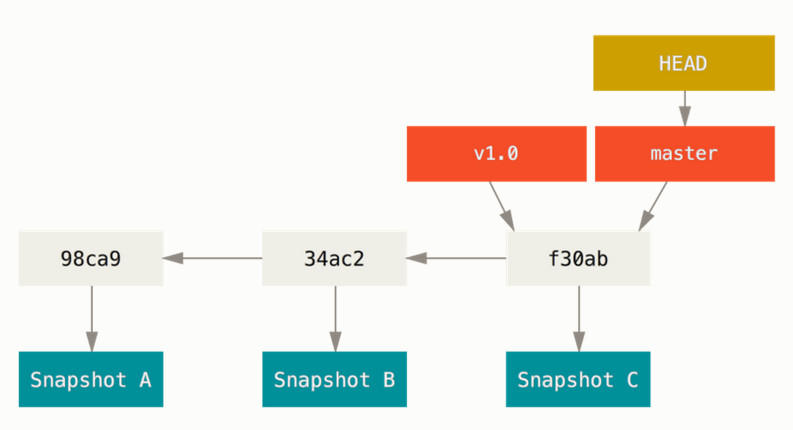
Although 'snapshots' are employed in the above explanations for clarity, the technical term is **commits**.

Branches

One of the powerful features of git is **branches**.

Branches are **movable pointers to commits**. Think of them as separate paths in code development, which you can later merge.

* By default, git creates a branch called main (formerly master) after running the git init command. **You should always keep it as your main branch.**
* HEAD (which we have encountered) is a pointer to the current location in the commit history.



**Using branches, one can**

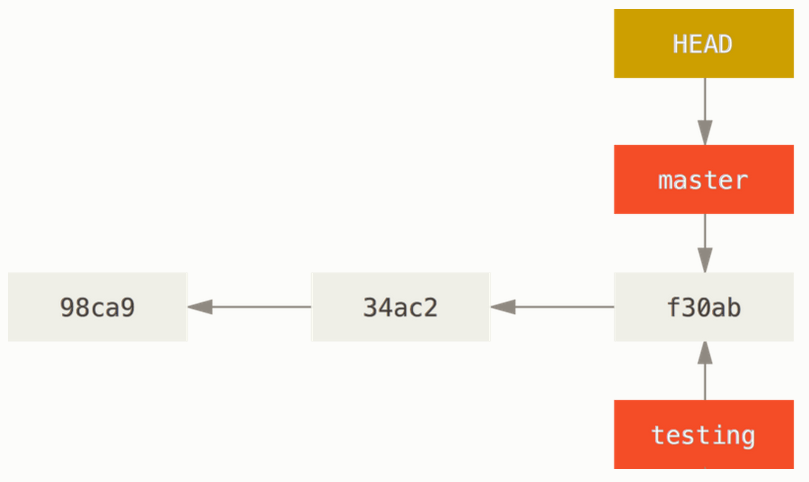
* work on new features separate from other developers.
* ensure that the entire process and workflow are structured and easy to follow
* test out experimental/work-in-progress (WIP) code without altering the **master branch**

Use branches **ALL THE TIME.**

Working with Branches

git branch NAME\_OF\_BRANCH is the basic command responsible for creating branches.

Below, we illustrate what happens after we run the git branch testing command.



Few things to note:

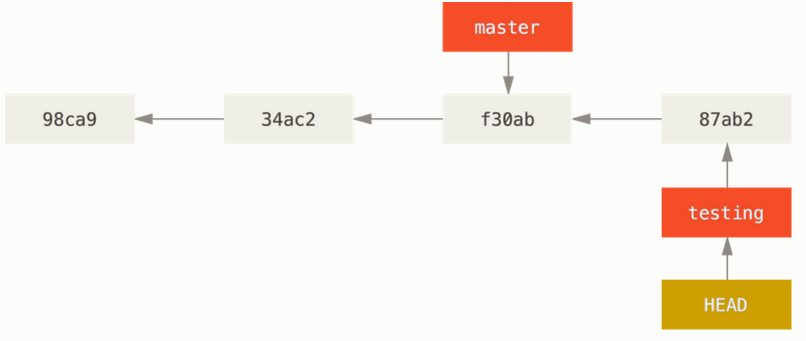
* **We are still on the master branch, as indicated by ~HEAD.**
* The new branch is merely a pointer to the last commit.

To switch to the new branch, we run the git checkout command:

git checkout testing

Tip: git checkout -b NAME\_OF\_BRANCH creates a new branch and checks it out (i.e. switches to the new branch) automatically.

Now that we are on the testing branch (HEAD points to it), we can perform the usual operations, including git add and git commit, and achieve results, as shown in the figure below:

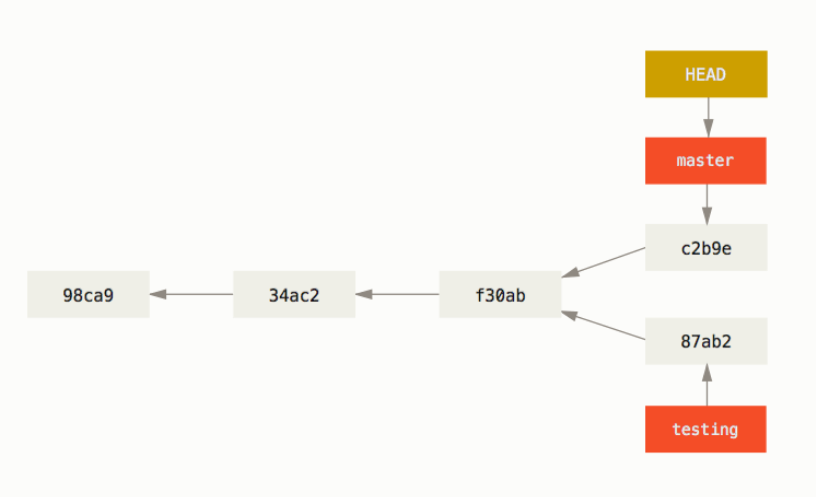


We can switch back to the master by simply running the git checkout master command.

Things to note:

* **Your local changes will revert to how they were on the master.**
* **This does not imply that your changes are lost. They are simply committed on another branch.**

Now, we commit on the master branch as well, which leaves us with the following (divergent) branch structure:



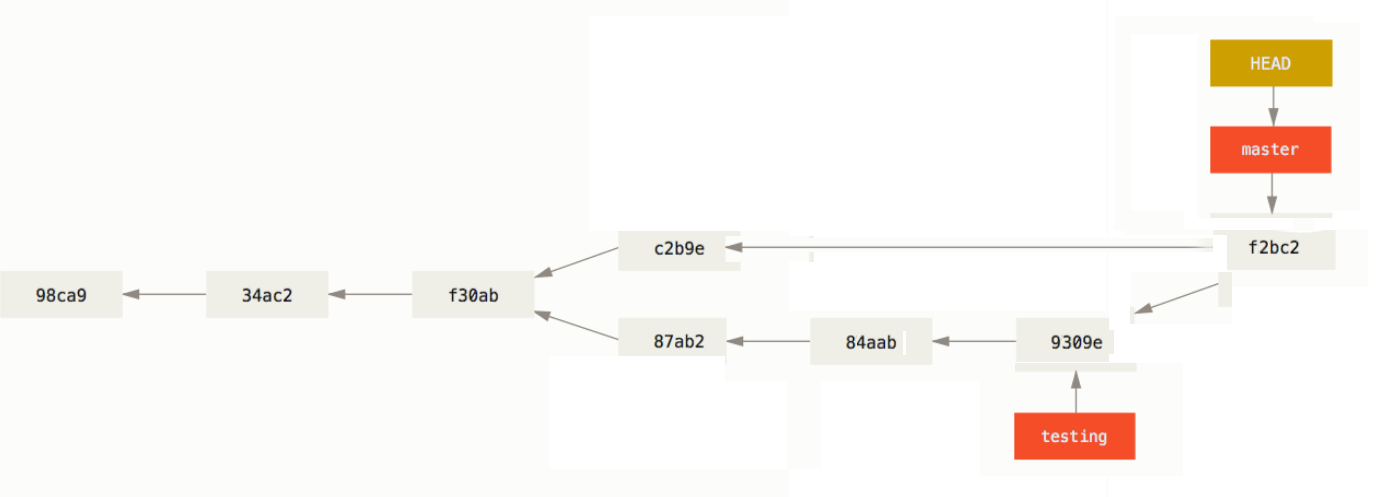
Tip

* **Pull all changes from the remote repository before creating a branch with new features.** This will minimise the risk of merge conflicts.

Merging

When you have completed your work on a branch, you can apply the changes to the main branch by merging it.

From the example above, let us assume that you have completed your work on the 'testing' branch. You can implement the changes from 'testing' into 'master' by merging.



There are three methods for merging a branch into another:

* merge commit
* squash and merge
* rebase and merge

In most cases, the default 'merge commit' will suffice. To obtain an in-depth understanding of the differences between these merging methods, read the following StackOverflow [thread](https://stackoverflow.com/questions/2427238/what-is-the-difference-between-merge-squash-and-rebase).

Exercise

Admittedly, that was a lot to process. Now, we put into practice all that we have learnt by creating a local repository, adding files to it, and finally committing those files. Thereafter, we will experiment with branches to improve your understanding of their working mechanisms.

Please carry out the following tasks and observe the changes in your local machine.

1. Create a new directory on your Desktop named 'AiCore\_git'.
   * Navigate to your Desktop from the terminal using the cd command.
   * Create the directory using the mkdir command.
2. Change your working directory to 'AiCore\_git'.
   * Once more, use the cd command.
3. Run git init to create a repository.
   * This will create a hidden directory that contains all the information regarding your commits.
4. List the files contained in 'AiCore\_git'.
   * Use the ls -a command to display all the files, including the hidden ones.
   * Notice that a directory named .git has been created.
5. Create two different files, e.g. 'test\_1.txt' and "test\_2.txt'.
   * Use the echo or the touch command for this task.
6. Check the status of the directory.
   * Run git status.
   * Read the message, and attempt to understand the state of your files.
7. Move the files to the **staged** state.
   * Use the git add command, followed by the name of a file to be staged.
   * Alternatively, you can stage all the files using the git add . command.
8. Check the status of the directory again.
   * Rerun git status.
   * What differences do you see with respect to the output of the previous git status command?
9. Take a snapshot of your new files so that git remembers them. In other words, make a commit.
   * Use the git commit command to commit all the files in the staged state.
   * Remember to add a commit message. Add the -m flag to the command, followed by your desired message in quotes.
     + For example, git commit -m "First commit".
10. Once more, check the status of your directory, and observe the differences.

Using branches

Here, we experiment with branches.

1. In 'AiCore\_git', create a new branch named 'testing'.
   * Use git checkout -b testing.
   * Here is a breakdown of the command syntax:
     + git checkout: switches to a different branch
     + -b: creates a new branch
     + testing: refers to the name of the new branch
   * Basically, we are creating a new branch called 'testing' and switching to it immediately.
2. Check the active branches in your directory.
   * Use git branch and see the output.
3. Create a new file named 'test\_3.txt'.
4. Stage and commit 'test\_3.txt'.
5. Switch to the main branch.
   * Use git checkout with the name of your main branch (Conventionally, it is either main or master).
6. List all the files contained in the directory.
   * If done correctly, 'test\_3.txt' should be out of sight.
   * However, do not be alarmed; 'test\_3.txt' is stored on the 'testing' branch. Note that none of the changes made on the testing branch were applied on the main branch. This is why it appears to have vanished.
7. Merge 'testing' into the main branch.
   * Use git merge testing.
8. Once more, list the files contained in this directory.
   * Great, we can see that 'test\_3.txt' is now in the main branch.

As you can now tell, using branches is a great way to not compromise your main code base.

Reverting Changes

If you accidentally add too many files and commit in a hurry, you can easily revert to the 'pre-changes' state.

For that, we can use the git reset command.

* git reset HEAD~ (the HEAD is actually written, it is not a placeholder here): reverts the last git commit and unstages (reverts git add) the files (you have to run git add to stage them again). Other than these, **no change will be made to the files. Therefore, you may rest assured that the files WILL NOT be deleted.**
* git reset [FILE]: reverts git add; if FILE is specified, it unstages the file; without any arguments, it unstages everything.

Resources

* [Pro Git Book](https://git-scm.com/book/en/v2) is one of the best resources on git (it is also a reference for some of the information provided herein).

GitHub

In this section, we present an overview of a good central server for storing code, i.e. **GitHub.**

**Before we proceed, please create a new GitHub account**[**here**](https://github.com/AI-Core/Content-Public/blob/main/units/Essentials/1.%20Git%20%26%20GitHub/1.%20Github/www.github.com)**.**

Creating a Repository

To create a repository (repo), please follow the instructions below.

* Click on the '+' sign on the top-right side of any page, and select 'New Repository'.

You will be redirected to the repository-creation page, which should display the following:

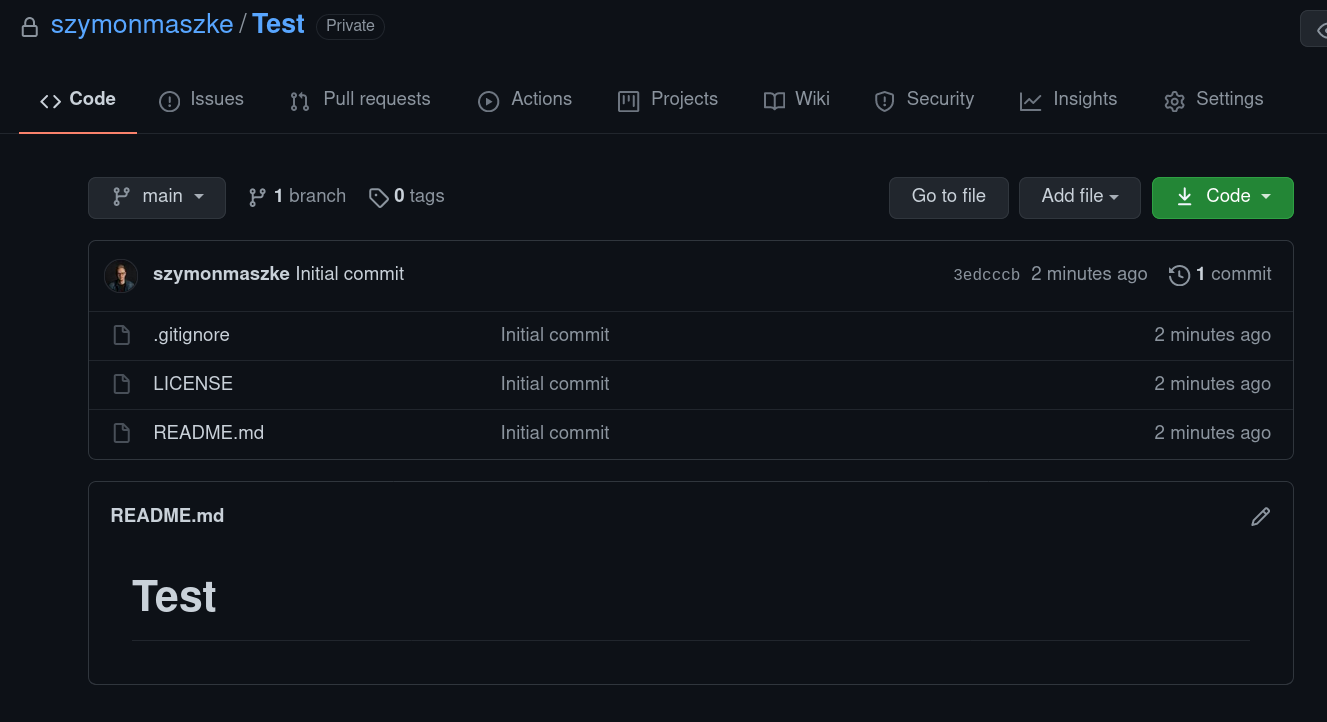
* **Public vs Private**: This determines whether the new repo is visible and accessible to everyone (forks and contributions are allowed) or just you and those to whom you grant access.
* **Initializing the repository**:
  + **Always add .gitignore at this step for a specific language (see below).**
  + It is advised to choose a license (MIT is the license of choice; however, you should explore other options.
  + Additionally, you can create a README.md file at this stage; you can also add it later after cloning the repo.

The .gitignore File

.gitignore is a file that prevents the addition of language-related 'junk' (files that are a byproduct of running the project and are not necessary for the project) to git.

* Throughout this course, you should always use python's .gitignore.
* You can also add specific files or filepath expressions, including regex (such as data), to this file, one on each line.

Cloning a Repository



You can clone your repository or any other using the git clone command. First, on the repo main page (as shown in the figure above), click on the green button to the right labelled 'Code'. Thereafter, copy the URL link in the 'HTTPS' tab, and run the following command in your terminal:

git clone <URL you just copied>

This will create a local version of the repo in a folder with the same name as the repo on GitHub.

GitHub tips

* **The size limit for a repository is 2 GB.**
* **Store your large files elsewhere (e.g. AWS's S3).**
* **Use git lfs (**[**https://git-lfs.github.com/**](https://git-lfs.github.com/)**) for large files that do not change often.**

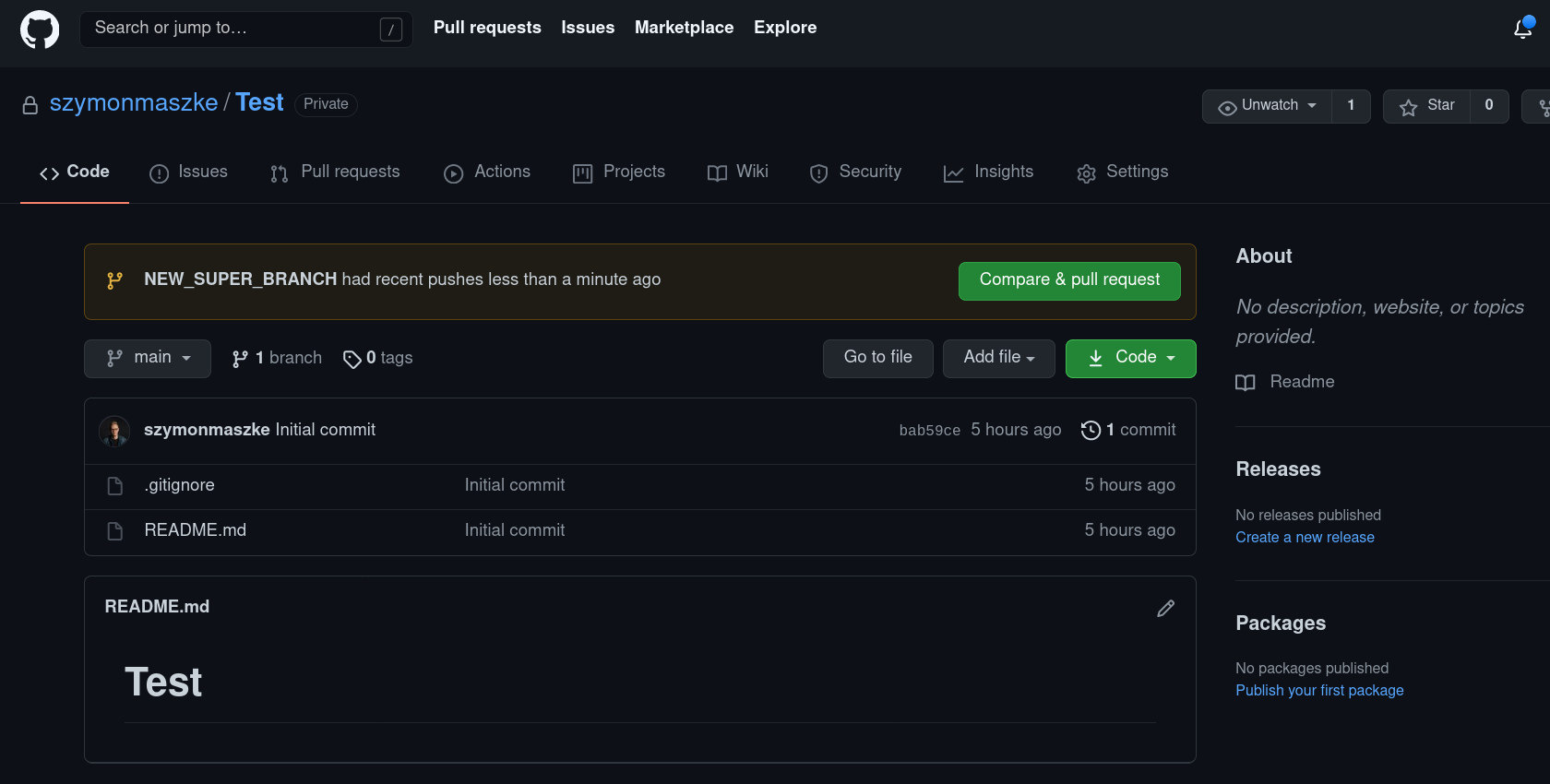
Pushing Changes to the Remote Repository

After committing, we can push our changes to the remote repository (outer server, **e.g. GitHub**).

git push -u origin BRANCH\_YOU\_ARE\_ON

In this course, you will always push your code to **your** GitHub repository.

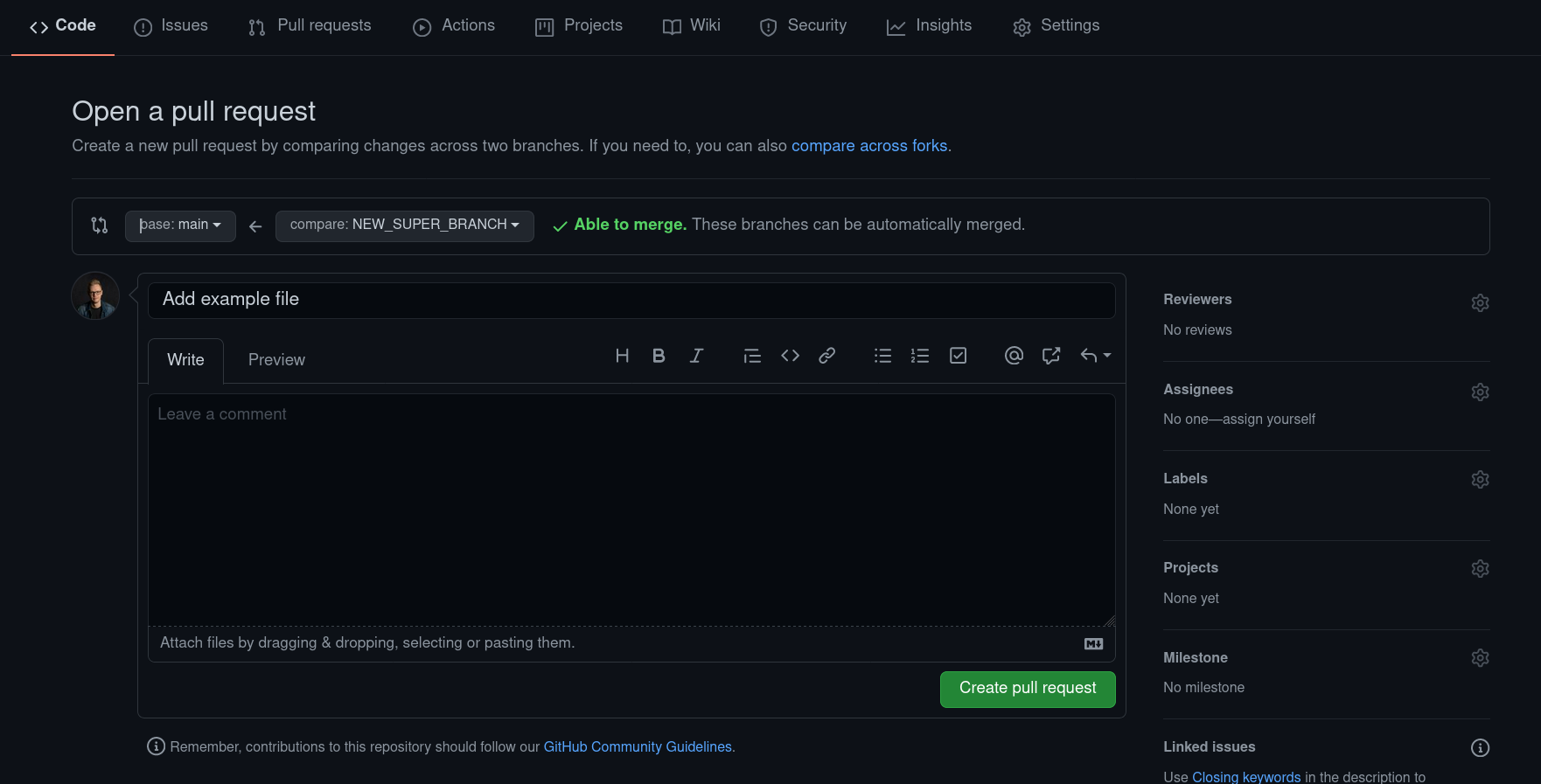
When you go to the main page of your remote repository, it should appear similar to the figure below:



By clicking on the wide green button, you will be making a pull request.

Pull Request

A pull request (PR) is made to the repository/project owner (or anyone with the appropriate rights) to **merge** changes located on one branch **upstream** (i.e. to the main branch, which is almost always master).



At this point, you can, amongst other things,

* assign someone to review your work (**do it all the time if you are collaborating on the project**)
* assign someone to work with you
* assign appropriate labels for clarity on which part of the project certain changes are related to (everyone should be aware at a glance).

**Most important is the wide green button for creating a PR.**

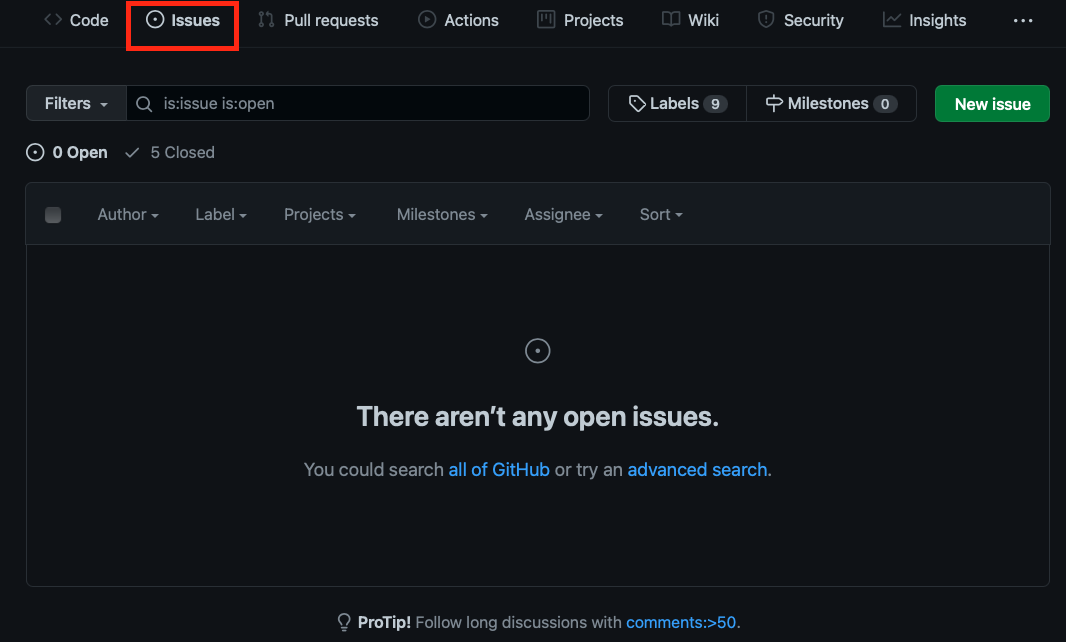
GitHub Issues

GitHub Issues is a tool for tracking the steps in developing your repo. It is quite helpful, particularly when working in a team and in need of a list of tasks to carry out.

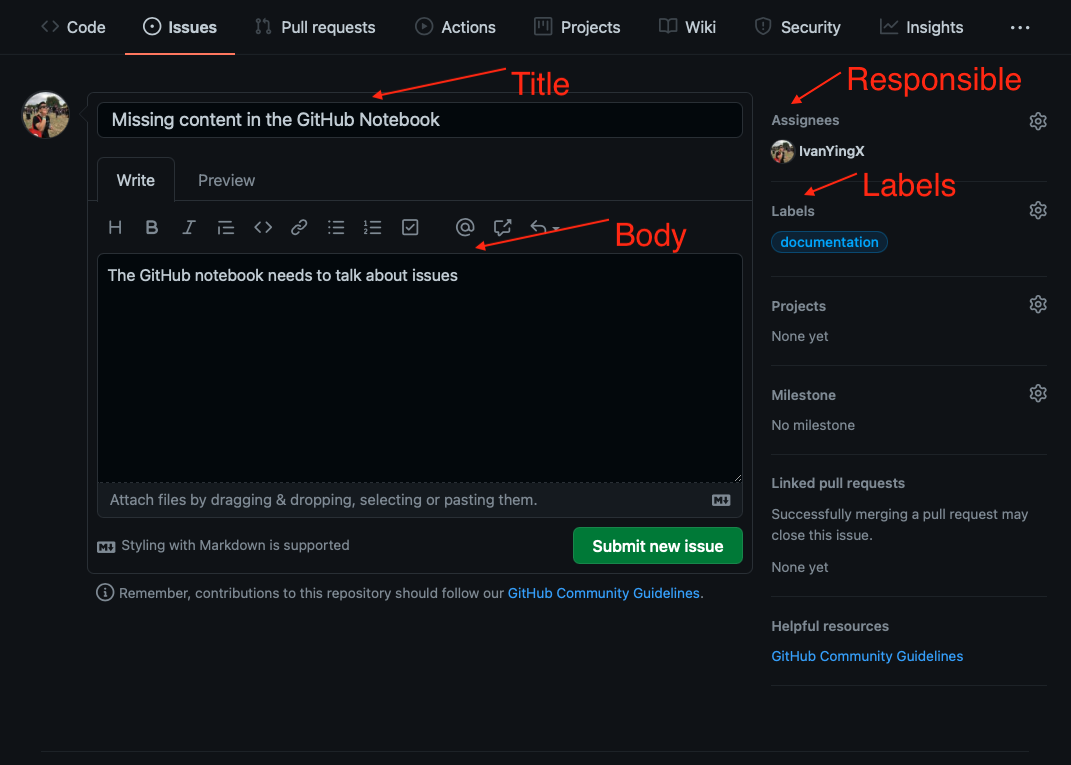
Each issue can be assigned a label/tag to define whether it is related to fixing a problem or adding new features... You can even create new tags.

To create issues, the following steps are required:

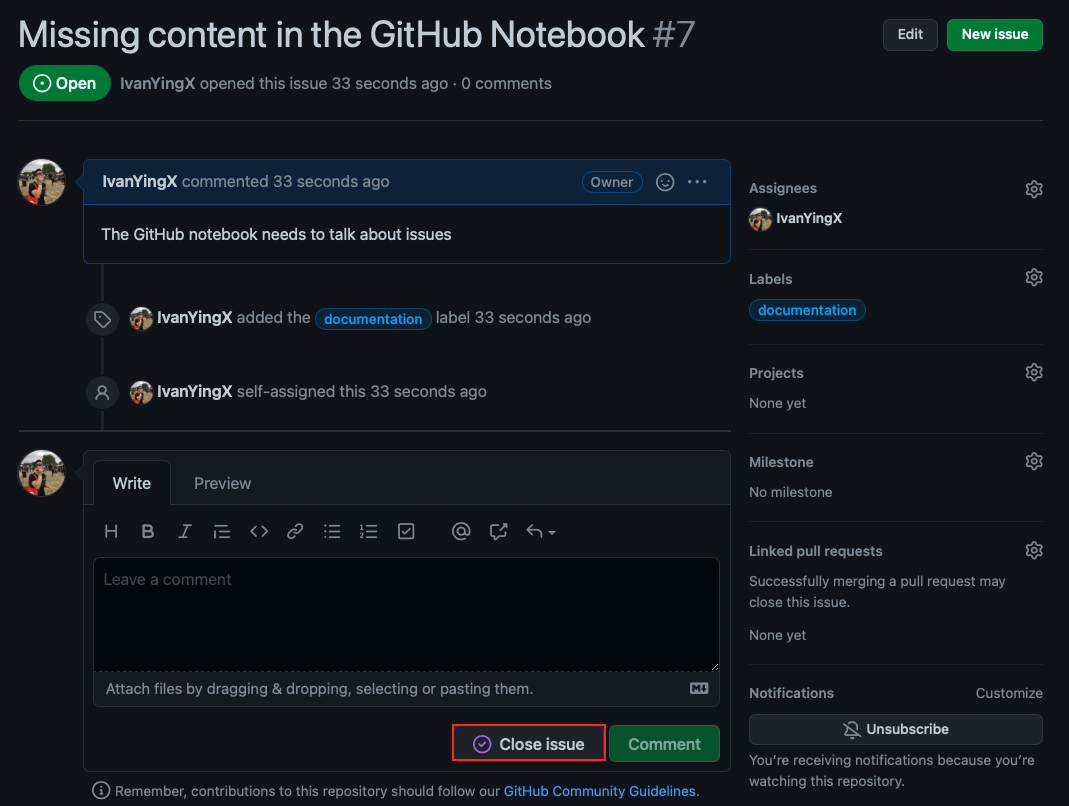
* Click on the 'Issues' tab on your repo page.



* Click on the green button labelled 'New issue'.
* In the next window, you can add a title and a description to your issue.
* Additionally, you can assign a tag to your issue, and allocate it to someone in your team for fixing.



* Upon completion, click on 'Submit new issue'. You will be redirected to a summary page of the issue. (Please note the issue number (Issue #7 in our case), as this information will come in handy shortly.
* At this point, we can close the issue by clicking on 'Close issue'.



However, it is preferable to use a keyword, followed by the number of the issue in a commit message. These keywords include

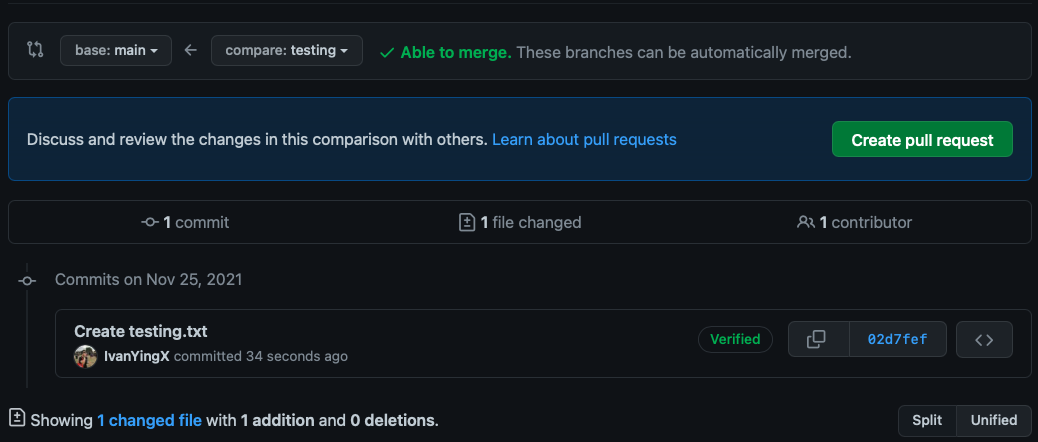
close, closes, closed, fix, fixes, fixed, resolve, resolves and resolved.

For example, if we make a commit and add the following message, 'Add issues content. This fix #7', the relevant issue (i.e. Issue #7) will be automatically closed. This is a pretty convenient function. Consider experimenting on this to attain a good grasp.

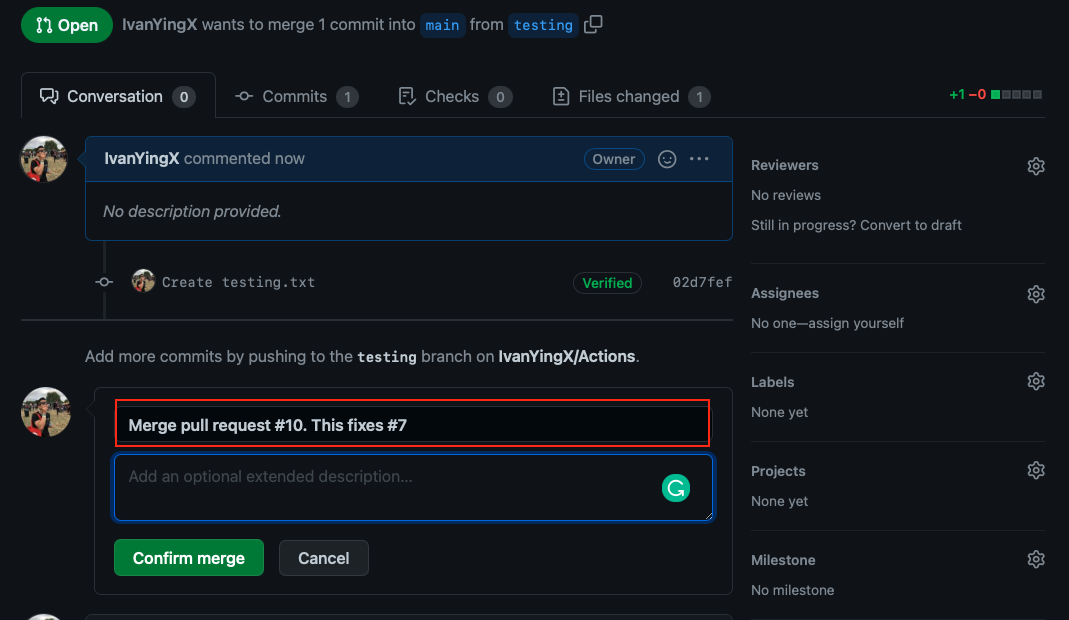
Exercise

Please follow these simple steps to set up your GitHub repo.

1. Create a new GitHub repo.
   * Give it a name of your choosing, such as 'Test'.
   * Tick the 'Add README' box.
2. Add an issue to your repo.
   * Do not focus on the name or the body.
   * Most likely, the issue will be Issue #1.
3. Clone the repo in your local machine.
   * Copy the URL of your repo following the directions provided earlier.
   * Run the git clone command in your terminal.
4. Change the directory to the cloned repo.
5. Create a new branch named 'testing', and switch to it.
   * Use the git checkout -b command.
6. Create a new file named 'test.txt'.
7. Stage and commit the changes.
8. Push the changes to GitHub.
   * Use the git push command.
9. Go back to GitHub, and inspect your branches.
10. Make a PR to merge the testing branch into the main.
    * Go to the PR tab.
    * Click on 'New Pull Request'.
    * Ensure that you are merging the testing branch into the main, as shown in the figure:



1. If all goes well, the 'Merge Pull Request' button should be green, allowing you to merge both branches with a message.
   * Use this message to close the issue you created earlier.
   * In our case, we closed issue #7. Please modify the issue number accordingly.
   * The message should include one of the keywords mentioned earlier, followed by the issue number.
   * An illustration of these steps is shown in the figure below.



1. Once you have clicked the 'Confirm Merge' button, inspect your issues.
   * Go to the issues tab.
   * Your issue should be gone.

Configuring Git Locally

To use most git features, you must provide your credentials. To do so, you can use the git config command, as follows:

* git config --global user.name 'Your Name'
* git config --global user.email 'your@email.com'

Ensure that you use the same email address you used when creating your GitHub account.

Resources

* [GitHub documentation](https://docs.github.com/en) - same as above, but for GitHub.

Numbers in Python

Learning Objectives

* Understand the difference between integers and floats.
* Understand the difference between None and int 0.
* Learn how to use Python as a calculator.

Numbers Types

* In Python, there are two types of numbers:
  + integers (int)
  + floating-point numbers (float)
* Basically,
  + an integer is 0 or a positive/negative whole number without a decimal point.
  + a float is any number with a decimal point.
* Due to the nature of floating-point calculations, floats are not exactly what you expect them to be; therefore, rounding is often applied.

Arithmetic Operations in Python

* The standard mathematical operators, i.e. **/, \*, + and −**, are applicable in Python.
* The order of operations is BIDMAS.
* The double asterisk (**\*\***) is utilised for powers (roots are simply fractional powers).
* Division always returns a float.
* There are a couple of special operations:
  + Modulo (x%y) outputs the remainder of dividing x by y.
  + Floor division (x//y) outputs the result of dividing x by y, rounded down to the nearest integer.

Note that there should be a single space around an arithmetic operator when in use, unless the addition of the space reduces clarity and readability (e.g. A + B, not A+B).

[ ]

# Addition  
2 + 1

3

[ ]

# Subtraction  
2 - 1

1

[ ]

# Multiplication  
2 \* 2

4

[ ]

# Division  
10 / 1

10.0

Floor Division

* This is occasionally referred to as integer division.
* It rounds down the result of a division operation to the nearest integer.
* It is performed using the double-backslash operator (//).

[ ]

7 // 4

1

[ ]

# Modulo  
7 % 4

3

[ ]

# If modulo 2 of a number is equal to zero, the number is an even number.  
6 % 2

0

[ ]

# Powers  
2 \*\* 3

8

[ ]

# roots can also be done this way.  
4 \*\* 0.5

2.0

[ ]

# Order of operations followed in Python  
(2 + 10) \* 10 + 3

123

[ ]

# Parentheses can be applied to specify orders.  
(2 + 10) \* (10 + 3)

156

[ ]

# Use round (expression, decimal\_places) to obtain a rounded result.  
round(10/3, 4)

3.3333

NoneType

* There is a clear distinction between None and 0.
* None has the data type, 'NoneType', and is, therefore, 'not a value' (it can be used as a placeholder before adding values).
* Zero (0) is an integer and, therefore, a value.
* The data type of a variable can be determined using the type() method:

[ ]

type(None)

NoneType

[ ]

type(0)

int

[ ]

# 0 + 1 works as they are both numbers.  
0 + 1

1

[ ]

# None + 1 throws an error, as None means there is nothing there.  
None + 1

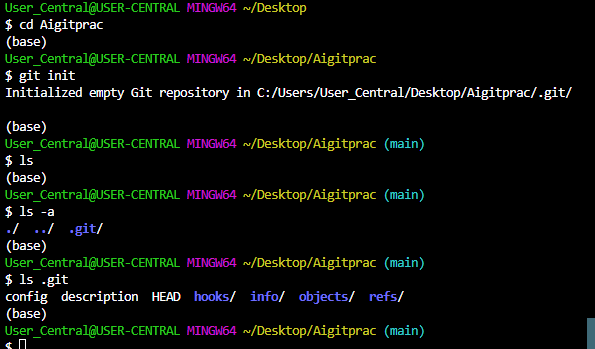
Conclusion

At this point, we should have a firm understanding of

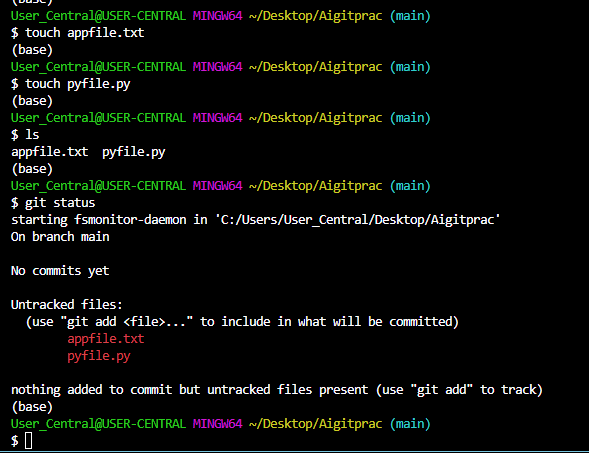
* ints and floats.
* None.
* how to use python as a calculator.
* how to format floats.

Further Reading

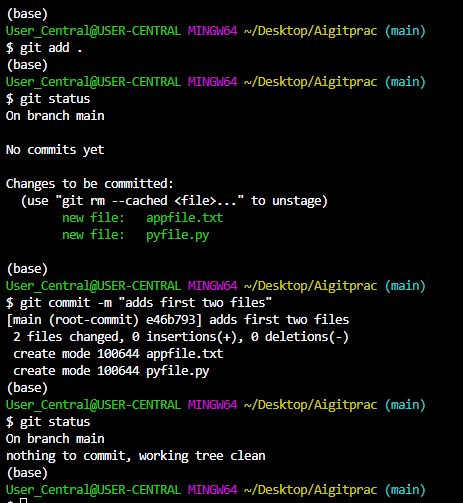
* Python None: <https://docs.python.org/3/c-api/none.html>



This directory repo stores all the info about the entire history of changes to your code. It also stores the current state of your code and the branch you are on



Files are in Modified state waiting to be staged (preparing files to be included in next commit) once we are happy with the changes



commited files using [*git commit –m “meassge”*] command.

Staged files using [*git add .* ] command.