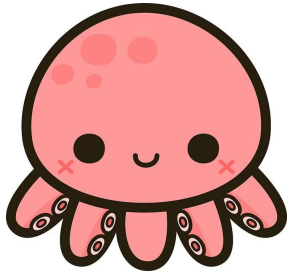


OCTOPUS

Onboarding & **C**ollaborative **T**utorial for
Open-source **P**rojects **U**tilised in **S**cience

What we'll cover here

What and how **git** and **github** works.

What is the **nu-ZOO**? And why should you care.

How to interact with **big collaborative repositories** (at least some of them).

- Making branches,
- adding changes,
- resolving issues,
- rebasing.

General style guide & tests.

A fun game to end the day.

What is git & github?

Git is “the information manager from hell”.

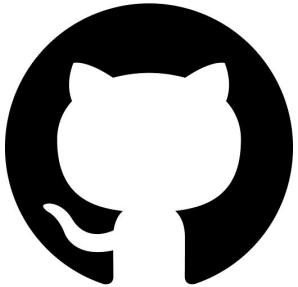
Source control software that allows:

- For the tracking and managing of changes to code over time,
- easy collaboration with fast branching and merging of many peoples work into one repository,
- the ability to explore project histories and revert to earlier versions.

Github is (generally) where these repositories are stored, and comes with lots of tools:

- ‘organisations’ and publicly available repositories,
- automatic testing suites,
- ‘forums’ for discussing issues, discussing changes, etc
- even more, Github and git have lots of features.

In practice, you’ll be working with both!



What is nu-ZOO?

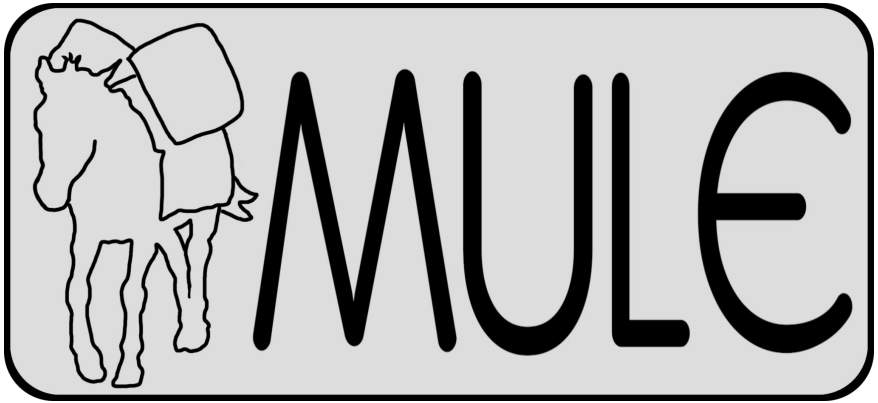
Organisation on github made by me and Brais to host any relevant working software repositories:

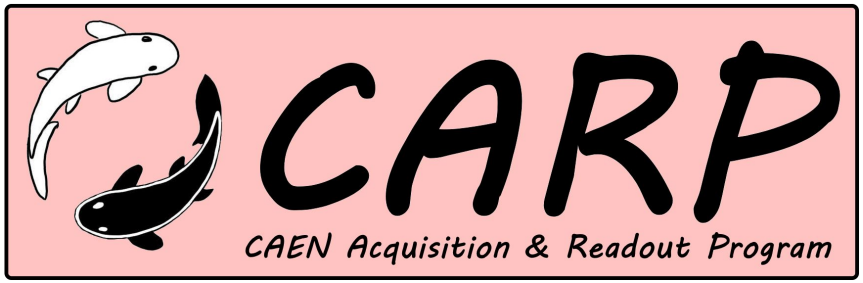
- MULE - **M**easurement and **U**tilisation of **L**ight **E**xperiments
  → decoding and analysis framework

- MARE - **M**ULE **A**rena for **R**ecursive **E**nhancement
  → Messy storage of work related to MULE

- CARP - **C**aen **A**cquisition and **R**eadout **P**rogram
  → WIP GUI-based software for working with CAEN digitisers.

- OCTOPUS - **O**nboarding & **C**ollaborative **T**utorial for **O**pen-source **P**rojects **U**sed in **S**cience.





Why work within nu-ZOO?

Developing good software development practices will make your life **easier** in the long term.

(It's also a very marketable skill).

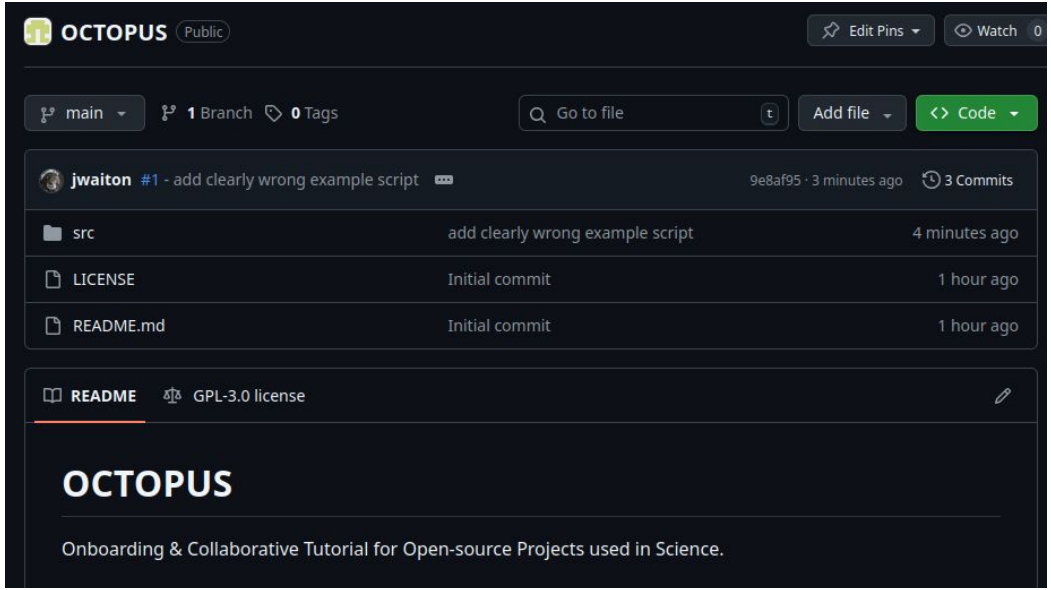
Stops duplication of the **near identical code**. If someone has written a baseline subtraction algorithm already, why waste two days writing a new one?*

Collaborating in shared frameworks allows for code to be more easily adapted and used for differing tasks (MULE).

Work through an example: OCTOPUS

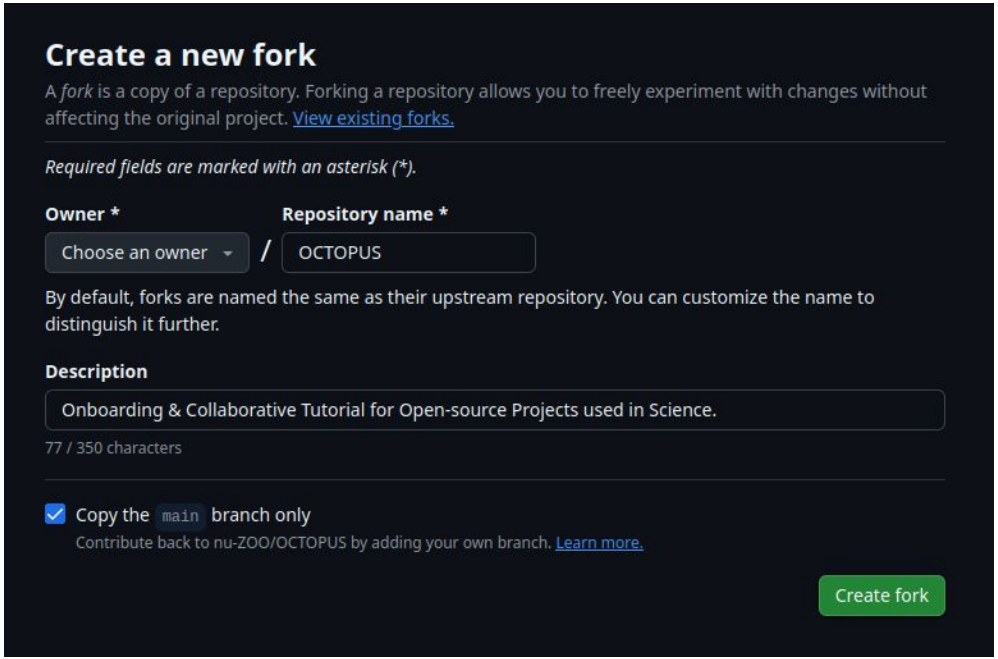
OCTOPUS - Onboarding & Collaborative Tutorial for Open-source Projects used in Science.

We'll work through a simple example that will go through all the steps here.



Work through an example: OCTOPUS

First, fork the repo:



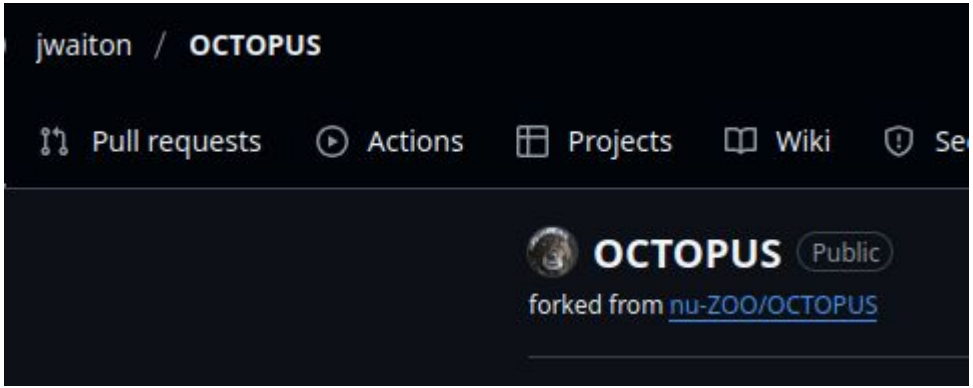
Work through an example: OCTOPUS

Once you have a fork (should look like this), you can pull this locally to work on it!

```
git clone https://github.com/jwaiton/OCTOPUS.git
```

This will be your ‘origin’

Whats an origin?

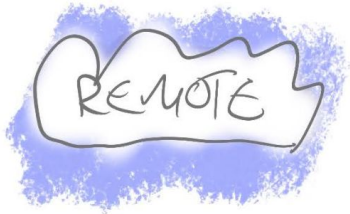


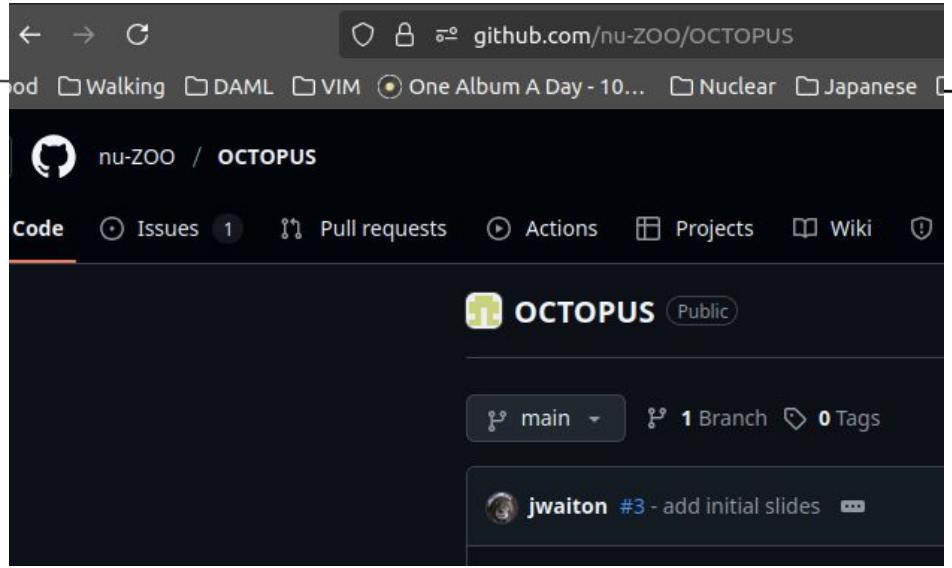
You have two ‘spaces’ to consider:

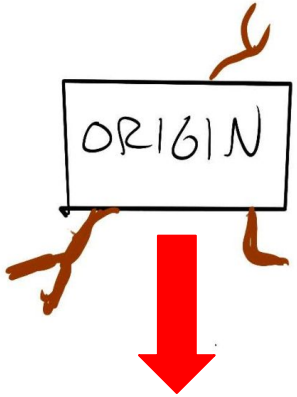
- **REMOTE**
  - This is everything online, github, gitlab, whatever you use.
- **LOCAL**
  - This refers to anything on your local machine (laptop, computer, etc)

UPSTREAM is the ‘main source’ repository for the code.

It should contain the version of the code you want people to work on top of.

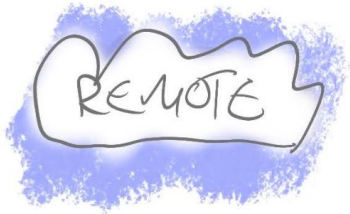


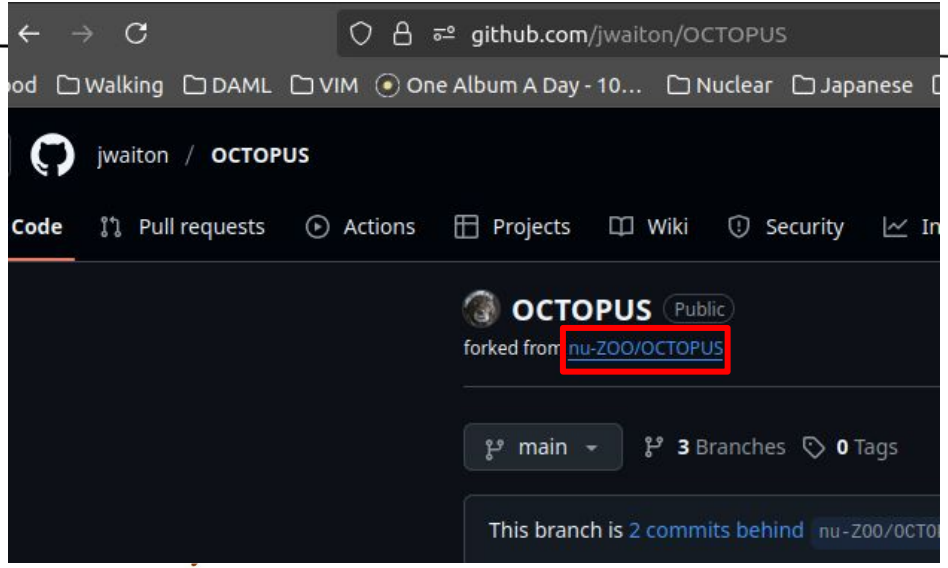


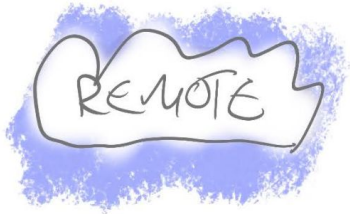


ORIGIN is your copy (or fork) of the upstream.

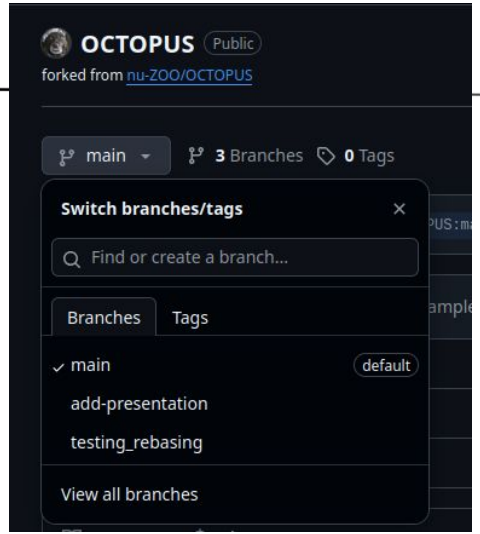
Most of your work should be implemented here (we'll come back to this)







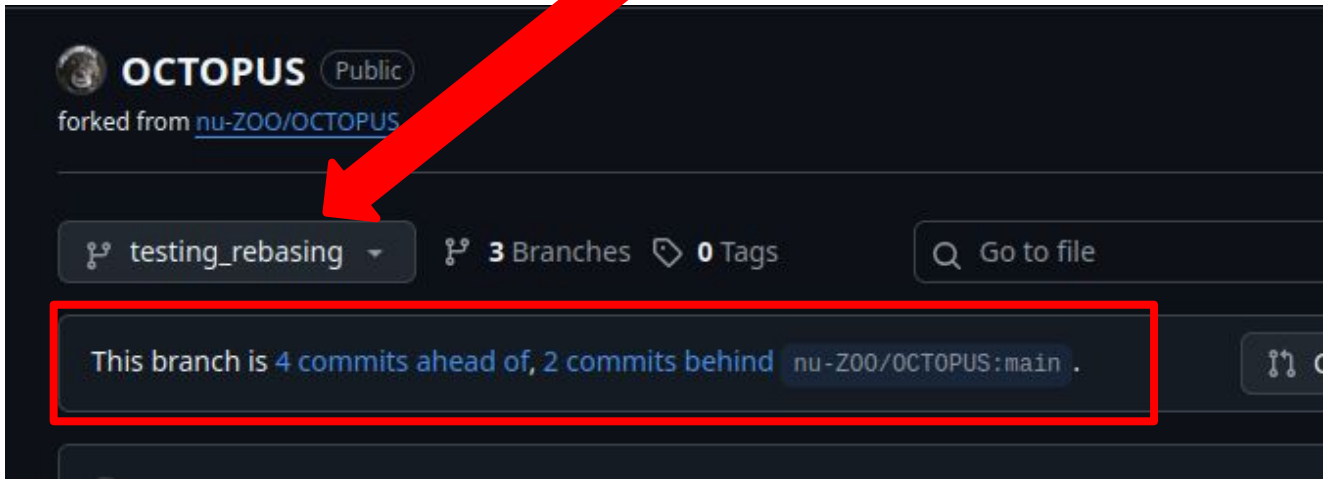
You can also work with other people's **ORIGINS**, they're generally called **REMOTES**

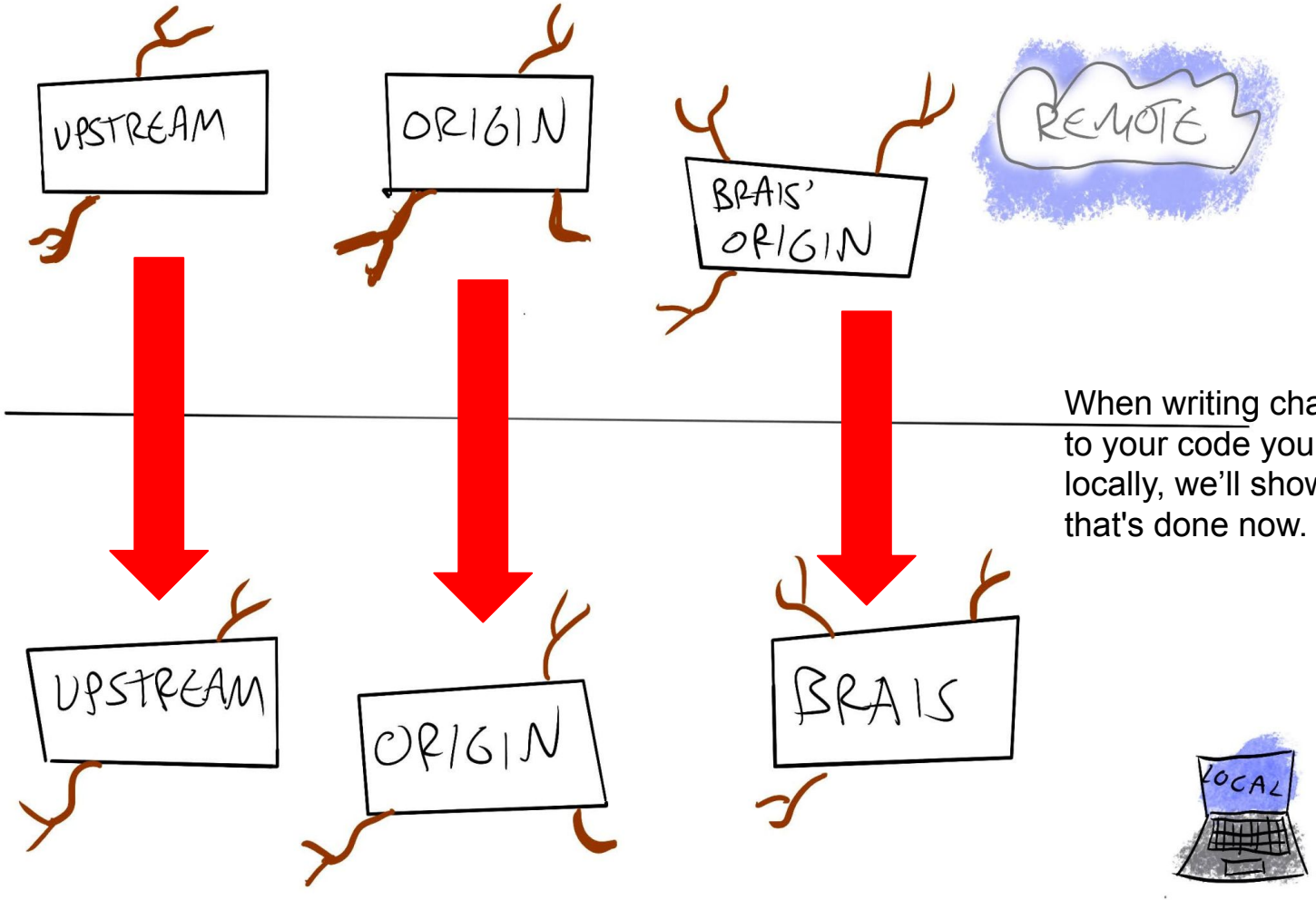


Within each repository, you have **BRANCHES.**

The **main** (or master) branch should match the main of the upstream. It is your "copy" of the source code.

Unique branches are where you should apply changes to the code.





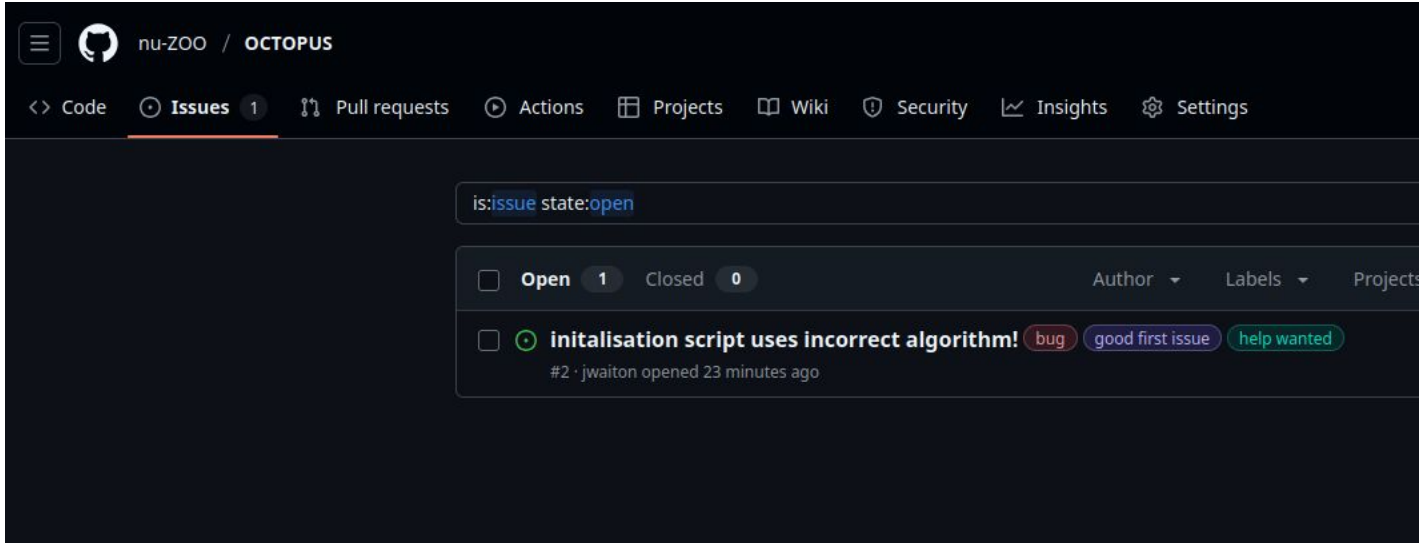
When writing changes to your code you do it locally, we'll show how that's done now.

Work through an example: OCTOPUS

Okay, you have your fork. Let's get to writing some code!

You can either create **new features**, or work on **issues**.

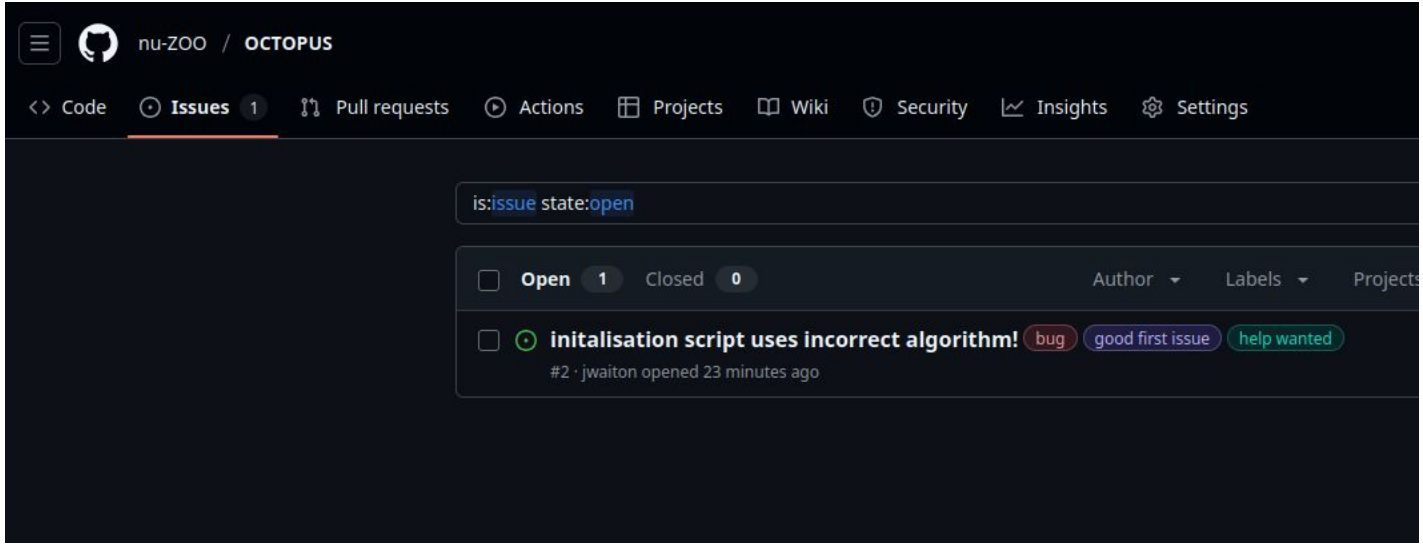
There already exists an issue!



Work through an example: OCTOPUS

To add different ‘remotes’, use

```
git remote add <name> <url git>
```



Work through an example: OCTOPUS

Lets resolve the issue, create a new branch to work in:

```
git checkout -b fix-incorrect-acronym
```

To look at the history (and where you are within it), use `git log`

```
e78368jw@e-10lux3072wzz:OCTOPUS$ git log
commit a8d74dc5e15cebaa392b3829ffa9db3b35054595 (HEAD -> main, origin/main, origin/HEAD)
Author: jwaiton <john.waiton@postgrad.manchester.ac.uk>
Date:   Mon Oct 6 15:24:13 2025 +0100

    add clearly wrong example script

commit 8b2ab96ce66fda11027c9a7bfb678e5f135c3a11 (upstream/main)
Author: John Waiton <john.waiton@postgrad.manchester.ac.uk>
Date:   Mon Oct 6 13:53:13 2025 +0100

    Initial commit
e78368jw@e-10lux3072wzz:OCTOPUS$
```

Work through an example: OCTOPUS

After making the changes, you can see what you've done with:

```
git status
```

and in more detail:

```
git diff FILE_NAME_HERE
```

```
e78368jw@e-10lux3072wzz:OCTOPUS$ git status
On branch main
Your branch is up-to-date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   src/example_code.py

no changes added to commit (use "git add" and/or "git commit -a")
e78368jw@e-10lux3072wzz:OCTOPUS$
```

Work through an example: OCTOPUS

To apply your changes:

```
git add FILE_AND_PATH_HERE
```

This ‘stages’ your file, prepared to be ‘committed’ like so

```
git commit -m ‘MESSAGE EXPLAINING WHAT YOUR COMMIT DOES’
```

You can commit multiple files at once.

Make your commits **imperative.** (for style)

Make your commits **small.** (for ease of reversion)

```
nothing to commit, working tree clean
e78368jw@e-10lux3072wzz:OCTOPUS$ git status
On branch main
Your branch is up-to-date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   src/example_code.py

no changes added to commit (use "git add" and/or "git commit -a")
e78368jw@e-10lux3072wzz:OCTOPUS$
```

Work through an example: OCTOPUS

To apply your changes:

```
git push
```

You may need to assign a ‘target’, but we get there when we get there.

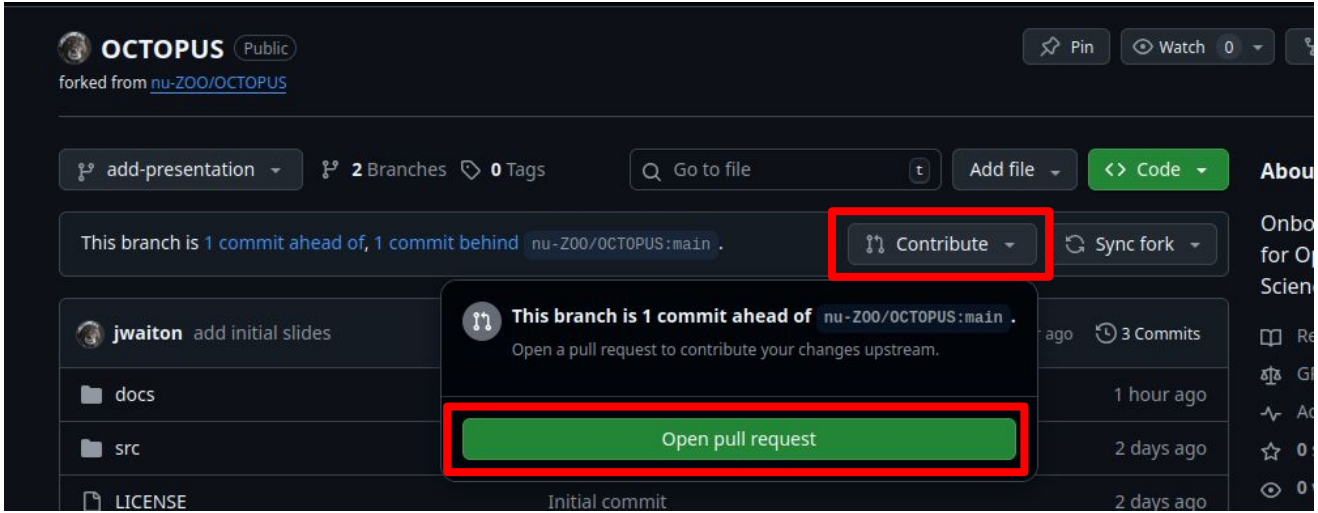
And your work should now be up on github!

```
nothing to commit, working tree clean
e78368jw@e-10lux3072wzz:OCTOPUS$ git status
On branch main
Your branch is up-to-date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   src/example_code.py

no changes added to commit (use "git add" and/or "git commit -a")
e78368jw@e-10lux3072wzz:OCTOPUS$
```

Work through an example: OCTOPUS

Your work should now be visible on your github in your branch!

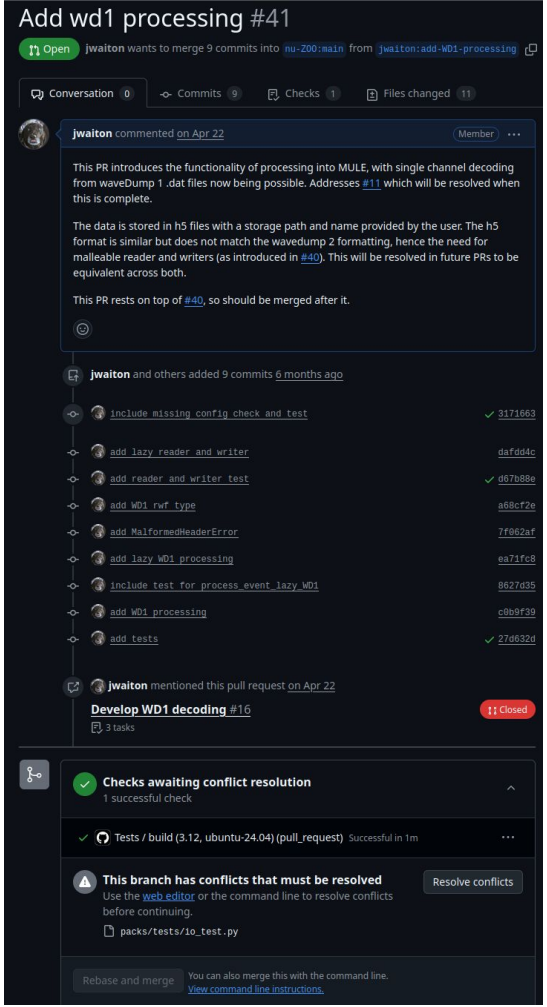
Once your work is ready to be added to the upstream, you can open a pull request.



Work through an example: OCTOPUS

Pull requests generally need to be reviewed to be accepted.

You should also write **tests** to demonstrate the functionality of your code.



Work through an example: OCTOPUS

Oh no! Your commits are out of sync with the main branch.

Do not worry! You can just **rebase**.

It's an incredibly powerful tool, but be a bit careful with it.

```
git rebase -i upstream/main
```

This will allow you to interactively rebase your prior commits onto the upstream. If you have conflicts, good luck 🫡

Work through an example: OCTOPUS

Other important commands

```
git fetch <name of remote here>
```

Downloads the repo (and branches)

```
nothing to commit, working tree clean
e78368jw@e-10lux3072wzz:OCTOPUS$ git status
On branch main
Your branch is up-to-date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   src/example_code.py

no changes added to commit (use "git add" and/or "git commit -a")
e78368jw@e-10lux3072wzz:OCTOPUS$
```

GAME TIME: Example

Lets say you make the test:

```
def test_addition():
    a = 1
    b = 1

    assert addition(a,b) == 2
```

GAME TIME: Example

Lets say you make the test:

```
def test_addition():
    a = 1
    b = 1

    assert addition(a,b) == 2
```

Your function could be in response:

```
def addition(a, b):
    return 2
```

You repeat this process until one side fails.

GAME TIME

You have been provided with two laptops for **two teams.**

Both laptops will have two branches available:

- **multiplication**
- **vowel_remover**

You need to commit changes and then pull the opposing teams changes to your machine as you work.

GAME TIME

You have been provided with two laptops for **two teams.**

```
def test_addition():
    a = 1
    b = 1

    assert addition(a,b) == 2
```

You will:

TEAM A1

- spend 5 minutes writing a test that checks if a function: **multiplication(a, b)** = a * b

```
def addition(a, b):
    return 2
```

TEAM B2

- spend 5 minutes writing a test that checks if a function: **vowel_remover(str)** removes vowels from a string

GAME TIME

You have been provided with two laptops for **two teams.**

You will:

TEAM A

- spend 5 minutes writing a function: **vowel_remover(str)** that passes the tests but doesn't do a * b

TEAM B

- spend 5 minutes writing a function: **multiplicastion(a, b)** that passes the tests but doesn't necessarily remove vowels from the string

GAME TIME

You have been provided with two laptops for **two teams.**

You will:

TEAM A

- spend 5 minutes writing another test that checks if a function: **multiplication(a, b)** = a * b

TEAM B

- spend 5 minutes writing another test that checks if a function: **vowel_remover(str)** removes vowels from a string

GAME TIME

You have been provided with two laptops for **two teams.**

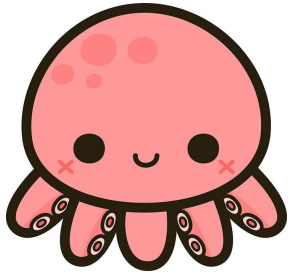
You will:

TEAM A

- spend 5 minutes modifying the function: **multiplication(a, b)** that passes the tests but doesn't do a * b

TEAM B

- spend 5 minutes modifying the function: **vowel_remover(str)** that passes the tests but doesn't necessarily remove vowels from the string



Thanks for listening!