

Git / bash shell Setup

- [UNSW: Mar 20-22, 2023 \(adacs-australia.github.io\)](https://adacs-australia.github.io)
- Bash shell download
 - Will install git along with it

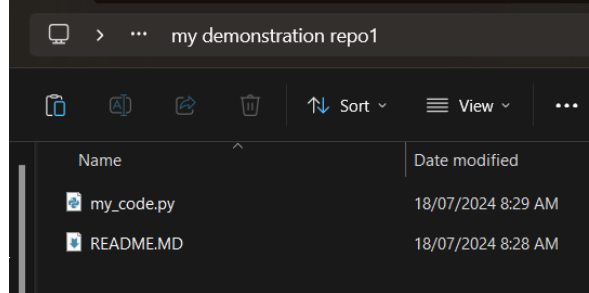
Creating repository for version control

- Enter the **directory that you want to version control for**, and open a bash terminal there. Inside, you will want to run the command “git init” to indicate that you want to do git version control for that directory.
- We track the changes made the directory through what we call “commits”. **Each commit is essentially a stored version of a file/files**. If we have multiple commits, we can use git to swap between them in order to look back on previous versions of a file/files. This saves us from having an overabundance of folders with different labels trying to manually keep track of different versions of these file/files. Let’s make our first commit. We will add one line to this commit.

```
git add <filename>  
git commit -m <message>
```

- We will add multiple files to the commit by using the “git add” command multiple times. If you want to add all files in a directory to your repo, you could use the command “**git add ***” before using
- Ensure that the **<message> term has no spaces in it**.

Example below



```
git init  
git add my_code.py  
git add README.MD  
git commit -m first_commit
```

- **Each commit made requires you to enter in the filenames you want to commit**
- If you alter any file and enter the same commands above (using a message like “updated_code” rather than “first_commit”), you store a more up to date version of your repo.

- Try making a new commit after modifying one of the files (e.g. my_code.py) in your repo

- If you enter the command “git log”, you can see all the commits (versions) made for your repo (eg below)

```
$ git log
commit 4b60572b4674bfd3e0353b81d6d448243d54e8bd (HEAD -> master)
Author: [REDACTED]
Date: Thu Jul 18 08:40:22 2024 +1000

    updated_code

commit e05b832daa55bb9e3c4d261eaccc73d13065189a
Author: [REDACTED]
Date: Thu Jul 18 08:34:27 2024 +1000

    first_commit
```

- Notice how each commit (version) has an ID number given above? We can use these to access previous versions of our files. Say we want to look at the version of my_code.py that was made in the first commit. We take the ID number of that commit, we take the filename, and we use a command “git checkout <id_number> <filename>”.

```
git checkout e05b832daa55bb9e3c4d261eaccc73d13065189a my_code.py
```

- If we now open my_code.py in our directory, we will see that it is now exactly as it appeared during the first commit. We can swap back to the newest version of the file by using the above command again, but with the id of the most recent commit. Make sure all your files are committed to your repo before you begin swapping to previous versions, or else you may lose some work
- It's easy to forget sometimes whether or not certain changes have been included in past git commits
- To check if anything in your repo has not been recently committed, you can use the command “git status”

Using .gitignore

- If the process of adding each file using git before a commit sounds tedious to you, you may want to use a .gitignore file. This file specifies the names of files/ subdirectories in your repo that you don't want included in a commit. Therefore, when you eventually do make a commit, all you need to do is run “git add .”, and all files in your repo, apart from the files specified by .gitignore, will be added to your commit. Below is an example of this

The screenshot shows a Windows terminal window and a .gitignore file editor. The terminal window is titled 'MINGW64/c/Users/kdror/Do...' and shows the following commands and output:

```
st 2024 version controll lesson/own test 2/.git/
MINGW64 ~/Documents/PhD year 1/c
cookies n code relevant classes/August 2024 version con
troll lesson/own test 2 (master)
$ git add .
MINGW64 ~/Documents/PhD year 1/c
cookies n code relevant classes/August 2024 version con
troll lesson/own test 2 (master)
$ git commit -m one
[master (root-commit) 43d87a7] one
2 files changed, 1 insertion(+)
create mode 100644 .gitignore
create mode 100644 file1.txt
MINGW64 ~/Documents/PhD year 1/c
cookies n code relevant classes/August 2024$
```

The .gitignore file editor is titled '.gitignore' and shows the following content:

```
more/**
```

- On the left, we have the contents of the repo: the subdirectory “more” contains a single .txt file. On the right, we have the contents of the .gitignore file, which specifies that we don’t want anything from “more” included in your commit. In the middle, we have a sample commit being made: you will see that no file from “more” was included in the commit, thanks to .gitignore
- Note that you will want a .gitignore file present upon your first commit, if you wish to use one.
- There’s a specific syntax that .gitignore files use in order to specify what files should be ignored: some of the specifics are explored on this website < [gitignore file - ignoring files in Git | Atlassian Git Tutorial](#)>
- We can do version control on our local computers alone, but we can move these repos to being stored online. We do this using an online repository storing website called GitHub
- The following links to a page discussing the linking of one’s github account to their local copy of git, as well as how to get a local git repo uploaded to github
- [Using GitHub – ADACS Coding Best Practices \(adacs-australia.github.io\)](#)
- If you have not already, you will need to copy your public SSH key to Github to allow your computer to have local repos connected to online repos ([Using GitHub – ADACS ECR Python Workshop \(adacs-australia.github.io\)](#))
- On github, we want to create a new empty remote repo that we can store our current local repo in: this will allow us to access our repo from anywhere.
- Upon creating this repo, we get some different commands we can use

Quick setup — if you've done this kind of thing before

[Set up in Desktop](#) or
 [HTTPS](#) [SSH](#)
<https://github.com/RoryRoryRoryRory/my-demonstration-repo1.git>

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```

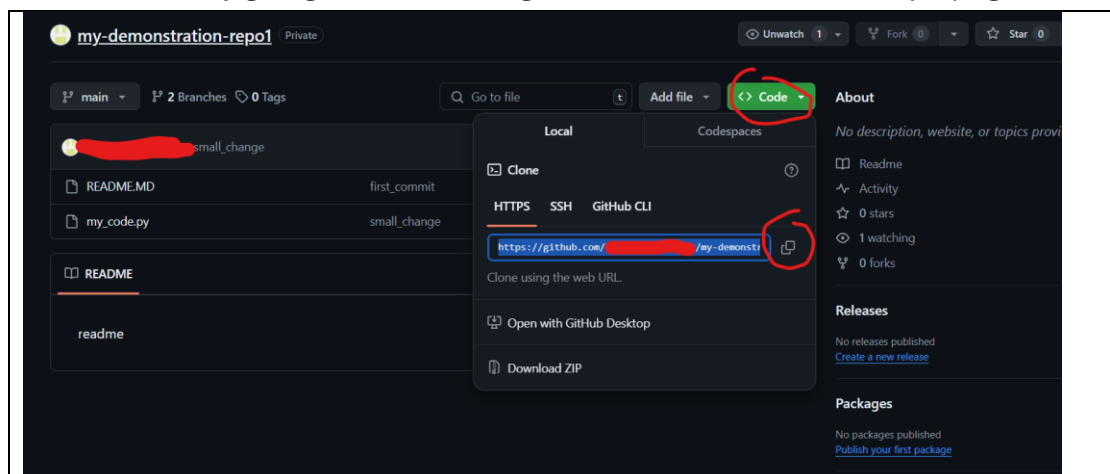
echo "# my-demonstration-repo1" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/RoryRoryRoryRory/my-demonstration-repo1.git
git push -u origin main
  
```

...or push an existing repository from the command line

```

git remote add origin https://github.com/RoryRoryRoryRory/my-demonstration-repo1.git
git branch -M main
git push -u origin main
  
```

- The 3rd option are the commands we will want to enter into our git command window to link our local repo to the new repo
- “origin”= a conventional name selected to refer to the local repo (this can be a different name if you wish)
- The word “main” refers to the main branch of development. When developing code collaboratively, you typically have many different branches of development. The branch command above creates the first main branch, simply known as “main”
- Use “git remote -v” to check that a link has been made.
- The advantage of having this online repo is that you can go ahead and download the repo onto any device/location you have.
- If we wanted to download a copy of our repo to another computer, we would open a bash terminal in the location we want to download a copy into, and we would run the command “git clone <GitHubURL>”. Note that we gain <GitHubURL> by going to the following subwindow in our online repo page.



- If you make any local commits, you can update the online repo to reflect these changes using the command “git push origin main”. This is what we call a push request.
- Say we have someone working on this code who would like to make commits to this online repo without altering the current “main” branch, as other people may still be using the main branch, and thus won’t want the code they’re working with changed in the meantime. For this, we are going to make a new branch of development.
- Let’s try making a new branch in our local repository called “feature-1”

```
git branch feature-1
git checkout feature-1
```

- Command 1 creates the branch, and command 2 swaps us over into that branch
- If we make changes to our repo and do a git commit, we now commit to this branch rather than our original “main” branch (try it out)
- We can update our repo with the details of this branch using “git push -u origin feature-1”
- It’s good to use branches in code development, as it means multiple people can work on code at the same time without all trying to make changes to the original branch of development at once.
- Imagine that we’re working in a group on some code, and you’re making changes on a specific branch, If the other members in your team are ok with it, you can make all of the changes you’ve made in your side branch appear in your main branch. To do this, we use the following commands

```
git checkout main
git merge feature-1
```

- Using “git log” confirms that our main branch now contains commits carried over from feature-1
- We can then commit these changes to the online repo using “git push -u origin main”.
- Note differences in using “merge” and “rebase” commands

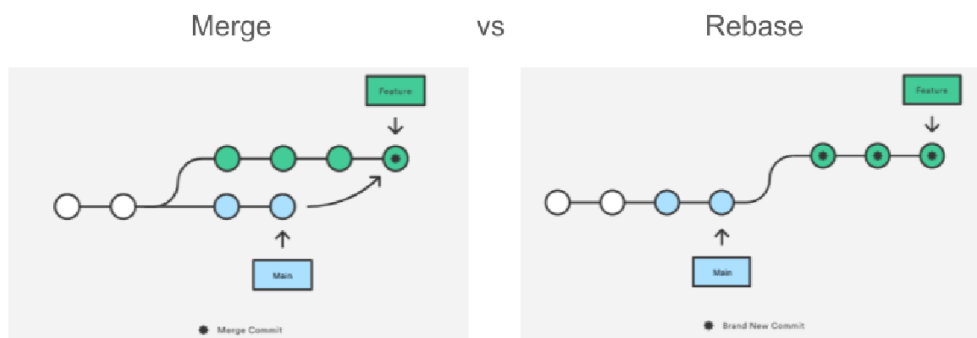


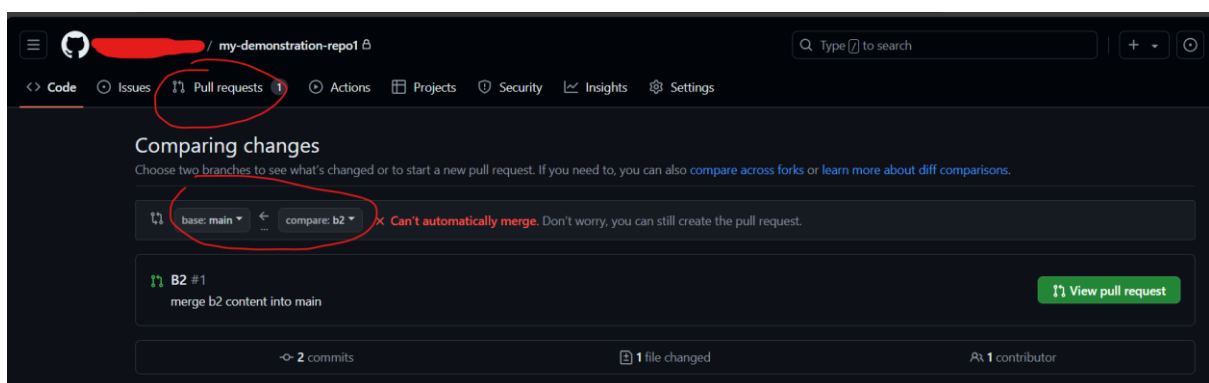
Image source: <https://www.atlassian.com/git/tutorials/merging-vs-rebasing>

- If you use `rebase`, it alters the commit history of the main branch to include all commits from the side branch. If you use “merge” the combination of code from the main and side branch is performed in a single new commit in the main branch. Rebase may cause issues if you are writing code collaboratively though, as pulling a rebased repo into your local repo could cause some strange conflicts due to differences in the commit history of both.
- You can’t rebase a branch if the main has updated since the branch has been created

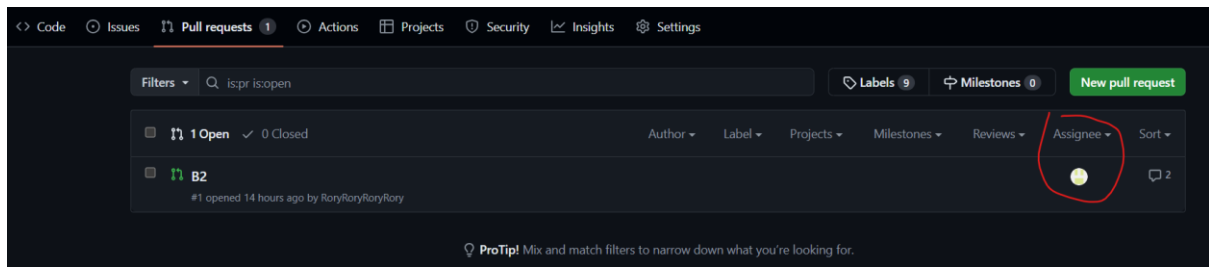
Say that we have a side branch (“b1”) that we want to merge with our “main” branch on github. Say that the both main and b1 have commits made, which change shared lines of code to something different.

“my_code.py” in the branch “b1”	“my_code.py” in the branch “main”
<pre> 26 27 "branch detail 1" 28 29 "branch detail 2" 30 31 outflow_dict={} 32 33 # outflow_dict["example_galaxy"]={ 34 # "CRPIX2":, 35 # "central row dist from CRPIX2":, 36 # "tc_results_dir":, 37 # "vertical_distance_resolution_in_kpc":, 38 # "masks": {}, 39 # } 40 </pre>	<pre> 26 27 "main detail 1" 28 29 "main detail 2" 30 31 outflow_dict={} 32 33 # outflow_dict["example_galaxy"]={ 34 # "CRPIX2":, 35 # "central row dist from CRPIX2":, 36 # "tc_results_dir":, 37 # "vertical_distance_resolution_in_kpc":, 38 # "masks": {}, 39 # } 40 </pre>

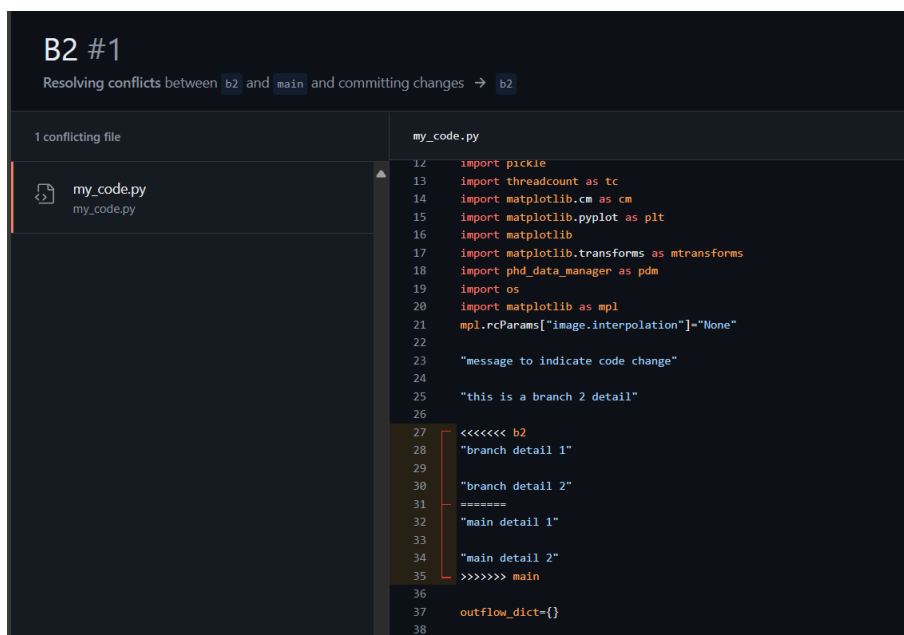
If we try merging these in github (or locally in the main branch using “`git merge b1`”), we will be unable to due to these conflicts. We need to perform what’s known as a pull request in order to resolve conflicts.



We go into the tab named “pull request”, we click the “new pull request” button, and we do to a new page where we specify (above) that we want to merge details of our “b2” branch into our “main” branch.



Once, the pull request is submitted, make yourself the assignee (the assignee is the one tasked with reviewing the conflict and deciding how it should be managed). If we click on the pull request's name, we should be taken to a page where we can find a button labelled “resolve conflicts”. If we click on that, we're taken to a page that looks like the following.



Each conflict in the code between the two branches is highlighted with this red line structure: this structure specifies differences between the two branches we're working with. To resolve the differences, we use this page to edit the code directly- we resolve conflicts by deleting everything they contain except for the code we want to keep. For example, if we prefer b2 over main, we delete all the lines marked below.

B2 #1

Resolving conflicts between `b2` and `main` and committing changes → `b2`

1 conflicting file

my_code.py

my_code.py

my_code.py

```

12 import pickle
13 import threadcount as tc
14 import matplotlib.cm as cm
15 import matplotlib.pyplot as plt
16 import matplotlib
17 import matplotlib.transforms as mtransforms
18 import phd_data_manager as pdm
19 import os
20 import matplotlib as mpl
21 mpl.rcParams["image.interpolation"]="None"
22
23 "message to indicate code change"
24
25 "this is a branch 2 detail"
26
27 <<<<<<< b2
28 "branch detail 1"
29
30 "branch detail 2"
31
32 =====
33 "main detail 1"
34
35 "main detail 2"
36 >>>>>>> main
37
38 outflow_dict={}

```

Once all conflicts are resolved, github will allow you to perform the merge. If we have no further use for the b1 branch, we could delete it locally using “git branch -D b1”. Use lower case d for a regular delete, and upper case D for deleting branch irrespective of whether its currently undergoing a merge or not. You can also delete the branch on github if you type the command “git push origin --delete b1”

Another type of conflict can occur where we have multiple users managing a repo. We might have the case where you make commits to your local repo that diverge fom the commits made in the online repo (see example below). This could occur if other users have been updating the online repo before you got a change to use the “git push” command.

In the “main” branch online	In the “main” branch locally
<pre> 43 files_to_use="3x3bin" 44 display_MAPPINGSIII_model=False 45 saving_directory=r""+"\" 46 specific_galaxies=["ESO120_016", "UGC00903"] 47 map_limits={ 48 "Halpaha": [0.1,10], 49 "ratio": [-0.53,0.1], 50 } </pre>	<pre> 43 files_to_use="3x3bin" 44 display_MAPPINGSIII_model=False 45 saving_directory=r""+"\" 46 specific_galaxies=["ESO120_016", "UGC00903"] 47 map_limits={ 48 "Halpaha": [1,1.66], 49 "ratio": [-0.53,0.1], 50 } </pre>

So if we try to use “git pull origin main” to get the up-to-date online repo into our local repo, we get a message like this that reports a conflict.

- ```

$ git pull origin main
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 980 bytes | 89.00 KiB/s, done.
From https://github.com/[redacted]/my-demonstration-repo1
 * branch main -> FETCH_HEAD
 2521b90..3856ddb main -> origin/main
Auto-merging my_code.py
CONFLICT (content): Merge conflict in my_code.py
Automatic merge failed; fix conflicts and then commit the result.

```

Opening our local code, we see some syntax very similar to the last time we resolved a conflict

```

files_to_use="3x3bin"
display_MAPPINGSIII_model=False
saving_directory=r""+"\"
specific_galaxies=["ESO120_016", "UGC00903"]
map_limits={
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<<<<<< HEAD (Current Change)
 "Halpha": [1, 1.66],
=====
 "Halpha": [0.1, 10],
>>>>>> 3856ddb0bc16439eacb1471aa86c54c4232281eb (Incoming Change)
 "ratio": [-0.53, 0.1],
}

```

We essentially resolve the conflict the same way that we did last time. We can use “git status” to quickly check if all conflicts are resolved. If so, we can continue to make commits/ push to the online repo if we need.

- Now run the command “git add my\_code.py”, where “my\_code.py” refers to a file where conflicts were just resolved in. Do this for each file where conflicts have been resolved in.
- Now run “git status”. We should get a message indicating that all conflicts have been resolved.
- Then run the command “git commit”, and the merge should take place (we can confirm that the merge has happened by using the command “git log”)

The same kind of conflict we just resolved can occur if you are locally trying to do a “git merge” command to join two (local) branches together that contain conflicts. You can use the same steps that you used above to resolve this kind of conflict as well