Introduction to Software Repositories

Bryan Scott

LSSTC Data Science Fellowship Program Session 15

Harvard-Smithsonian Center for Astrophysics

July 2022

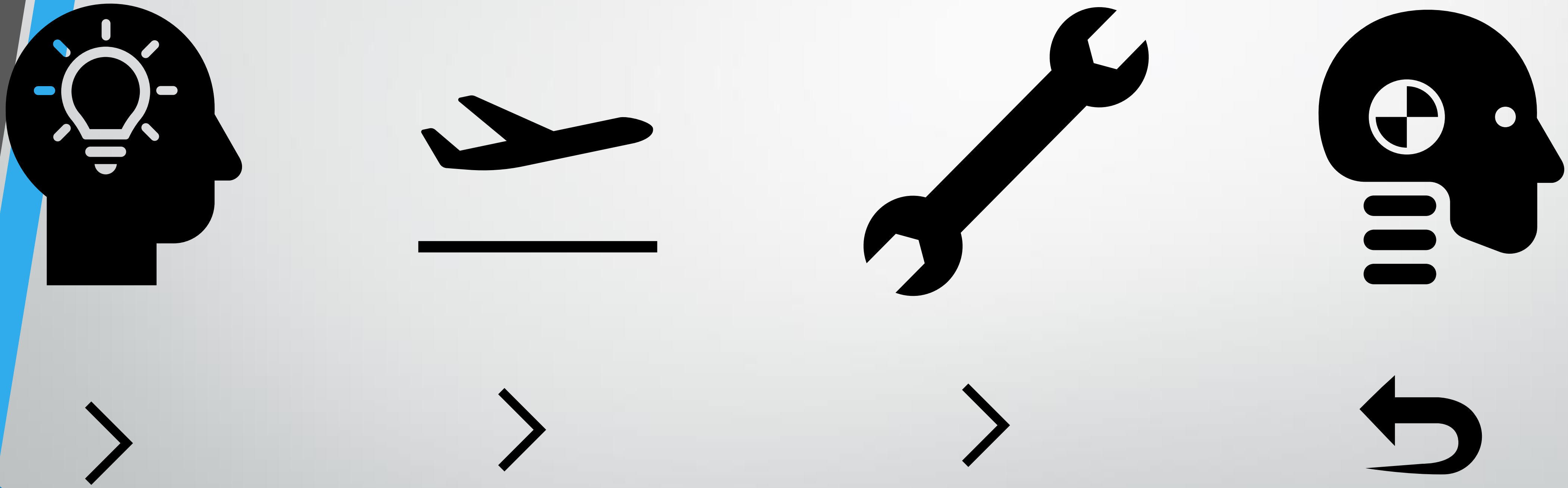
Learning Outcomes

- By the end of this talk, you should be able to implement a distributed version control workflow with git.
- And you should be able to use github to manage collaborative projects with multiple branches.

Literature this is based on: the main references for this talk are the Pro Git book available at git-scm.com/docs and *Beginning Git and GitHub: A Comprehensive Guide to Version Control, Project Management, and Teamwork for the New Developer* by Mariot Tsitoara

What makes learning git difficult?

Let's be real – development workflows...



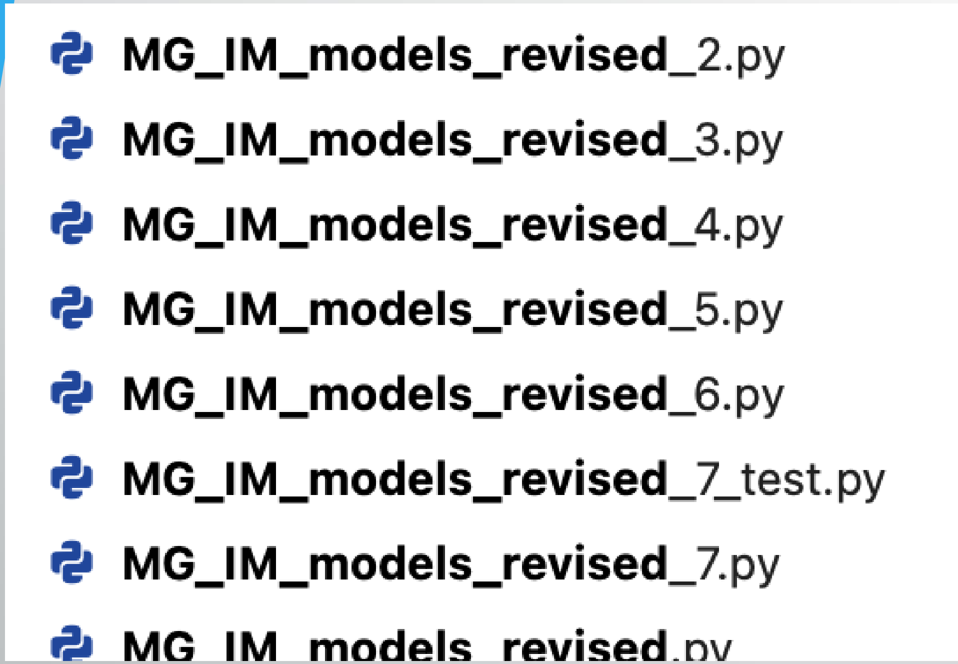
Brilliant idea!

Prototyping

Iterative Fixes

Oh no!

Problems in typical development workflows...



MG_IM_models_revised_2.py
MG_IM_models_revised_3.py
MG_IM_models_revised_4.py
MG_IM_models_revised_5.py
MG_IM_models_revised_6.py
MG_IM_models_revised_7_test.py
MG_IM_models_revised_7.py
MG_IM_models_revised.py

Paper_draft_final.tex
Paper_draft_final_revised.tex
Paper_draft_final_revised_final.tex
Paper_draft_final_revised_final_submission.tex
Paper_draft_final_revised_final_submission_with_comments.tex
Paper_draft_final_revised_final_submission_with_comments_and_responses_draft.tex
(and so on....)

What is version control?

- Version control is a *system* for creating a *reproducible* record of changes to a *project*.
- The key idea is that there is only **one** project. *No matter* how big or complicated that project is.

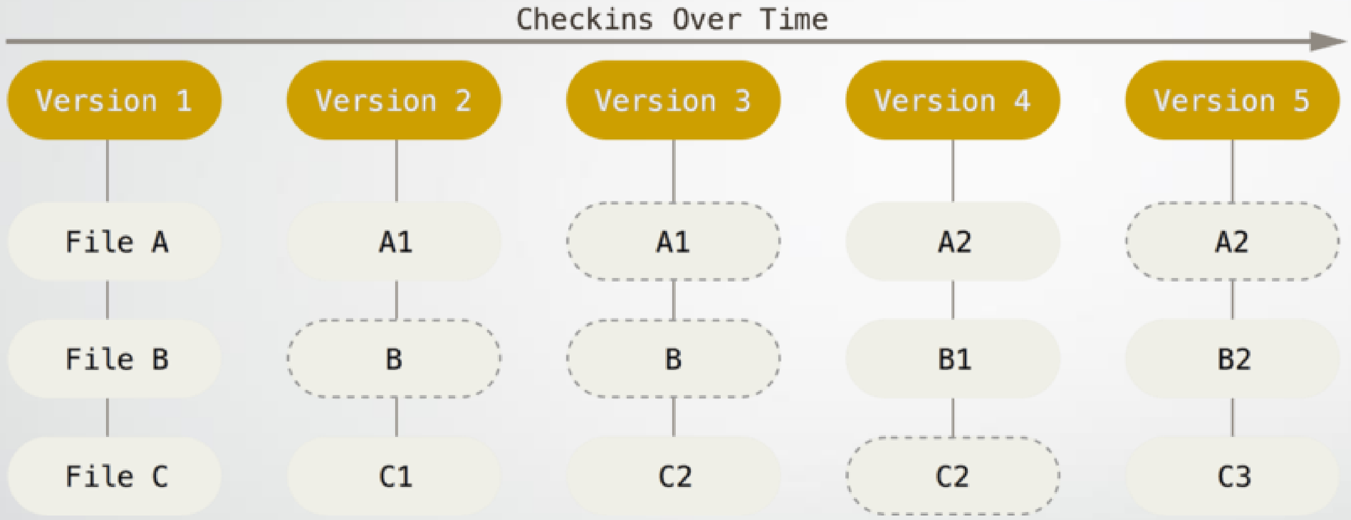
Introduction to git

Distributed Version Control

The Distributed Version Control Model (Git)

- A software repository is a place for storing software and metadata about it.
- In a distributed version control system, each developer has a copy of the repository – while it may be convenient to choose a "central" copy as a way of aiding collaboration – there is no authoritative version of the repository.
- Git is an example of a distributed version control system.

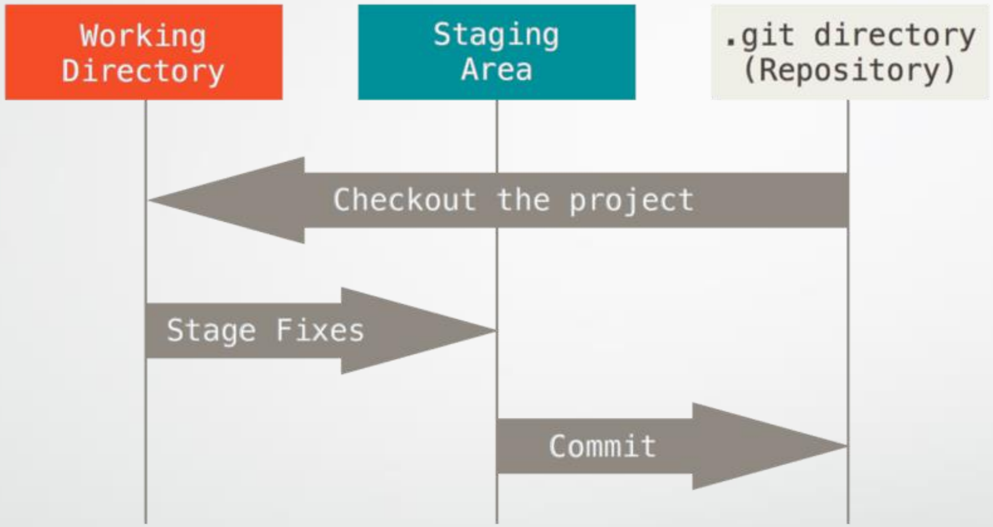
How does Git work?



Git stores snapshots of the project over time. If a new file is added or an old file changed, the new version will be stored in the next snapshot.

But if a file is unchanged, git simply links to the version stored in the previous snapshot.

Git States



Files can be in one of three states. *Modified, staged, and committed.* Files move between the three stages as you work. After editing a file it is *modified.* You then *stage* the file which tells git to include it in the next snapshot. You then *commit* the changes (save a snapshot of the project).

Creating and Initializing a Git Repository

Now create the repo by running the command,

```
git init
```

Creating and Initializing a Git Repository

Now create the repo by running the command,

```
git init
```

Creating and Initializing a Git Repository

Next we add the files in the working directory so they will be *tracked* by git.

```
git add *
```

We've just *staged* the files in this repository. Now we will add them to the repository by making a *commit*. Every commit requires a short description of what the commit includes.

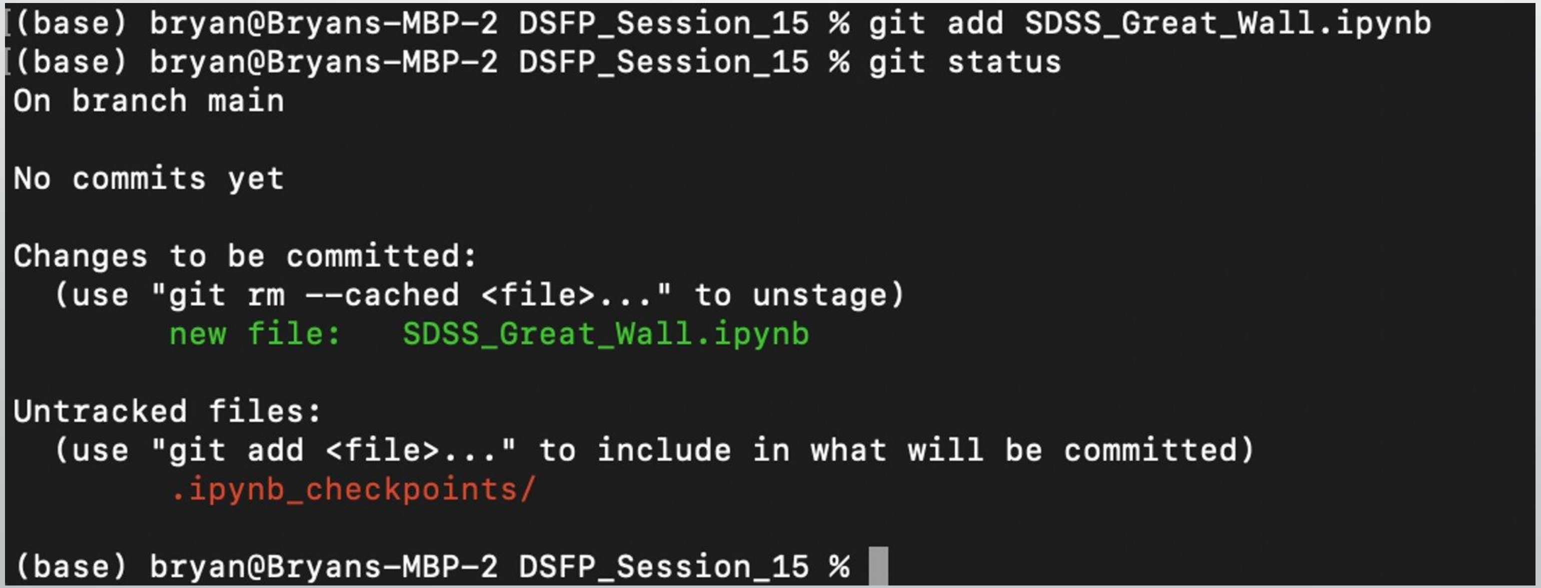
```
git commit –m 'Initial commit to initialize the repository.'
```

Good News everyone! We have our first software repository!

Git *Status* Command

- The git *status* command is your main tool for checking the current state of the repository

```
git status
```

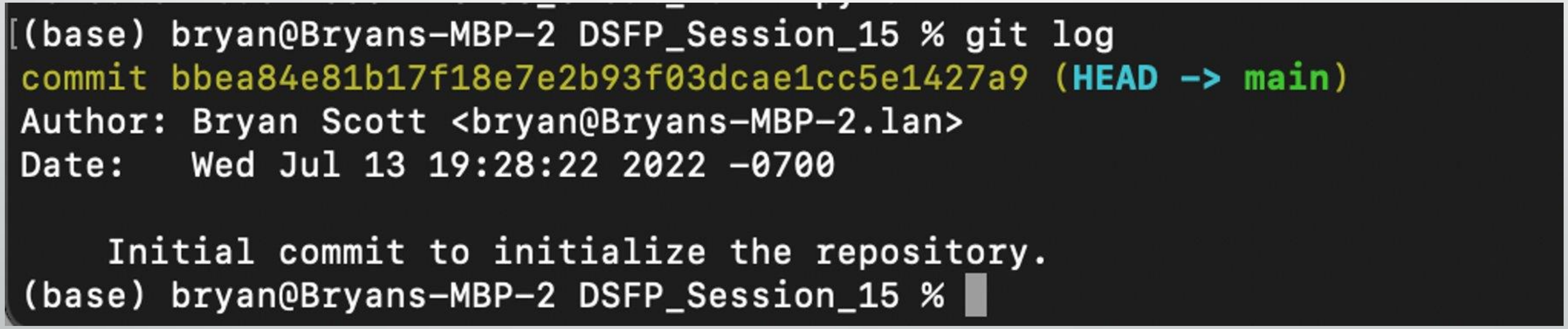


```
(base) bryan@Bryans-MBP-2 DSFP_Session_15 % git add SDSS_Great_Wall.ipynb
(base) bryan@Bryans-MBP-2 DSFP_Session_15 % git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   SDSS_Great_Wall.ipynb

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .ipynb_checkpoints/

(base) bryan@Bryans-MBP-2 DSFP_Session_15 %
```

Git *log* command

The git log command shows the commit history.

```
git log
```



```
[(base) bryan@Bryans-MBP-2 DSFP_Session_15 % git log
commit bbea84e81b17f18e7e2b93f03dcae1cc5e1427a9 (HEAD -> main)
Author: Bryan Scott <bryan@Bryans-MBP-2.lan>
Date:   Wed Jul 13 19:28:22 2022 -0700

    Initial commit to initialize the repository.
(base) bryan@Bryans-MBP-2 DSFP_Session_15 %
```

The usefulness of software repositories for different kinds of developers...

- For "small projects", tracking changes with git (or another VCS) and making regular commits makes your work more reproducible and gives you a "global undo" if you introduce a catastrophic error into your code.

- For "large projects", a distributed VCS can allow multiple developers to work independently and synchronously. Changes are merged later (and conflicts resolved appropriately).

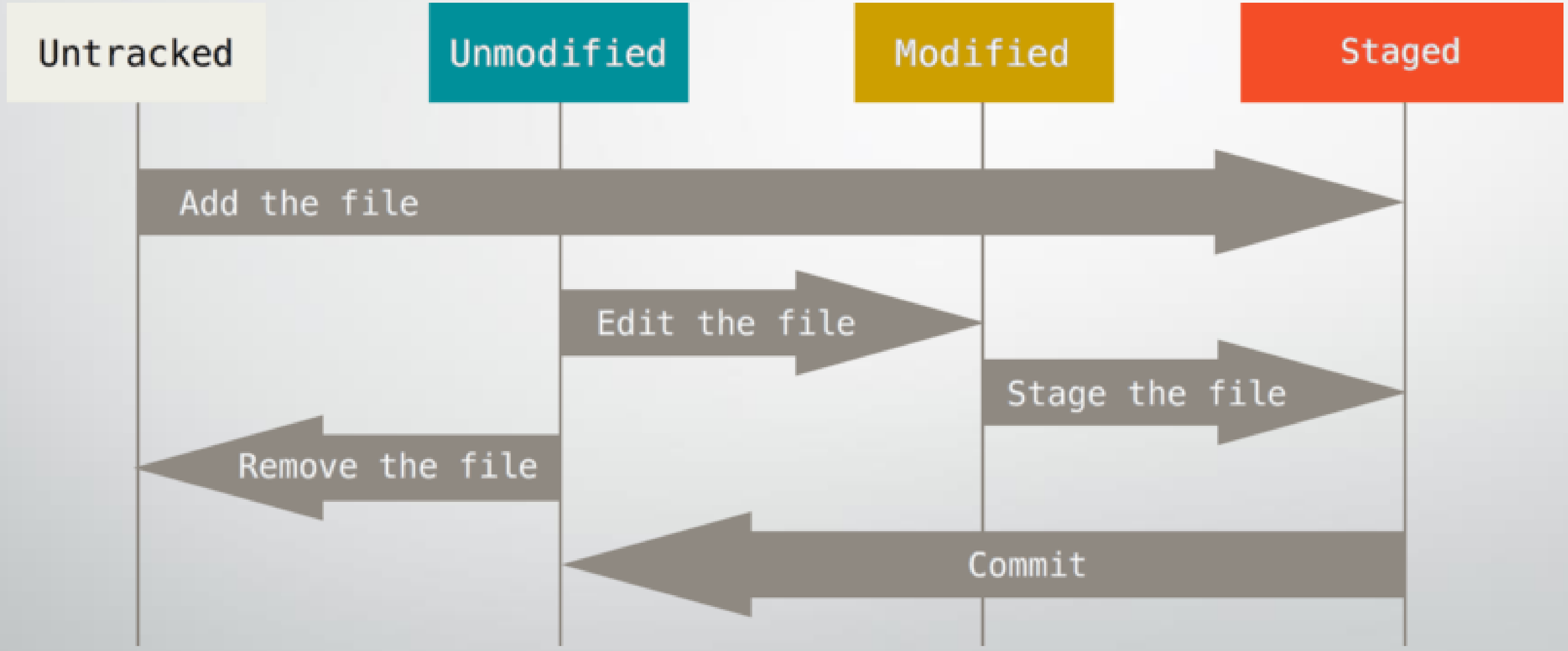
Git Usage – Changes to the Repository

- Files in your project can be in one of two states – *tracked* or *untracked*

- Git tracks all files that were in the last snapshot (*commit*) or that have been created and staged with the git *add* command.

- To add changed files to the repository, you need to stage them with *add* and then snapshot with *commit*

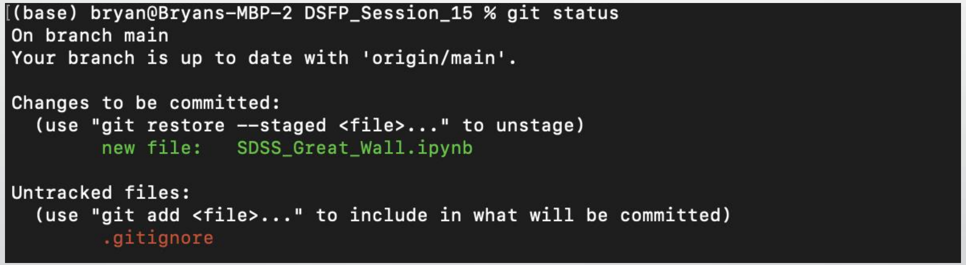
Git File Cycle



Commiting modified files

- Suppose we have made a change to a previously committed file. What does git status look like and how do we commit the changes?

```
git status
```



```
[(base) bryan@Bryans-MBP-2 DSFP_Session_15 % git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   SDSS_Great_Wall.ipynb

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
```

```
git add
```

A word of caution: you need to *add* **everytime** you make a change to a file. Git stages the file exactly as it was each time you run *add*.

Git ignore

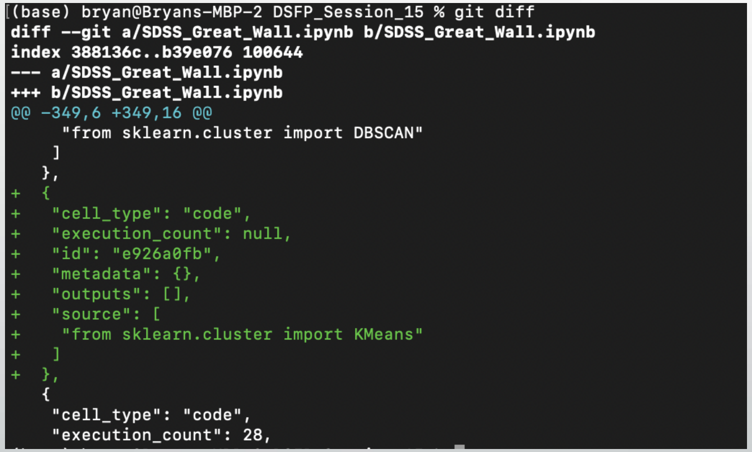
- To exclude files from git tracking, we can create a .gitignore file.

```
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf
```

Git diff

- Now we get to the powerful parts of git. The git diff command tells you exactly what has changed between the last time files were *staged* and what you have in your *working directory*.

```
git diff
```



```
(base) bryan@Bryans-MBP-2 DSFP_Session_15 % git diff
diff --git a/SDSS_Great_Wall.ipynb b/SDSS_Great_Wall.ipynb
index 388136c..b39e076 100644
--- a/SDSS_Great_Wall.ipynb
+++ b/SDSS_Great_Wall.ipynb
@@ -349,6 +349,16 @@
      "from sklearn.cluster import DBSCAN"
     ]
    },
+   {
+    "cell_type": "code",
+    "execution_count": null,
+    "id": "e926a0fb",
+    "metadata": {},
+    "outputs": [],
+    "source": [
+     "from sklearn.cluster import KMeans"
+    ]
+   },
    {
     "cell_type": "code",
     "execution_count": 28,
```

- Caution: git diff doesn't show *all* changes – only those that have **not** been staged.

More on Commits

- To save a snapshot of the project, *commit* your staged files to the repository

```
git commit
```

- This will open a text editor for you to add a one-line description of the changes made in this commit. These changes will be recorded in the git *log.*

- Two useful flags for commit are –v and –m, -v outputs the diff to the editor so you can see what changes are being included, and –m allows you to add the commit message inline.

You can also stage and commit at the same time with the –a flag.

Removing files

- To remove a file from the repository, you need to do the opposite of git add and then commit the changes. The command for this is git rm.

```
git rm [file]
```

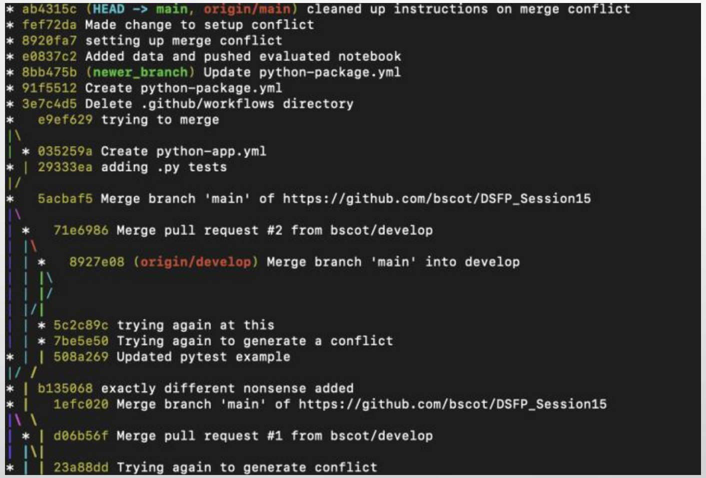
- If a file has already been staged or modified, you must force the operation with the –f flag – this prevents accidental deletion of a file that can't be recovered from a previous snapshot. Git is a powerful *recovery and reproducibility* tool.

- The –cached flag to git rm removes a file from tracking but leaves it on your hard drive.

Commit History

- The price of using git is a more complex workflow, but the payoff is that we now have a complete record of changes to our project. To view the record, we use the git log command:

```
git log –oneline --graph
```



```
* ab4315c (HEAD -> main, origin/main) cleaned up instructions on merge conflict
* fef72da Made change to setup conflict
* 8920fa7 setting up merge conflict
* e0837c2 Added data and pushed evaluated notebook
* 8bb475b (newer_branch) Update python-package.yml
* 91f5512 Create python-package.yml
* 3e7c4d5 Delete .github/workflows directory
*   e9ef629 trying to merge
|\
| * 035259a Create python-app.yml
* | 29333ea adding .py tests
|/
*   5acbaf5 Merge branch 'main' of https://github.com/bscot/DSFP_Session15
|\
| *   71e6986 Merge pull request #2 from bscot/develop
| |\
| | *   8927e08 (origin/develop) Merge branch 'main' into develop
| | |\
| | |/
| |/|
| | * 5c2c89c trying again at this
| | * 7be5e50 Trying again to generate a conflict
* | | 508a269 Updated pytest example
|/ /
* | b135068 exactly different nonsense added
* |   1efc020 Merge branch 'main' of https://github.com/bscot/DSFP_Session15
|\ \
| * | d06b56f Merge pull request #1 from bscot/develop
| |\|
* | | 23a88dd Trying again to generate conflict
```

- The git log is really a "Directed Acyclic Graph" - we will talk more about this when we talk about merges and branches.

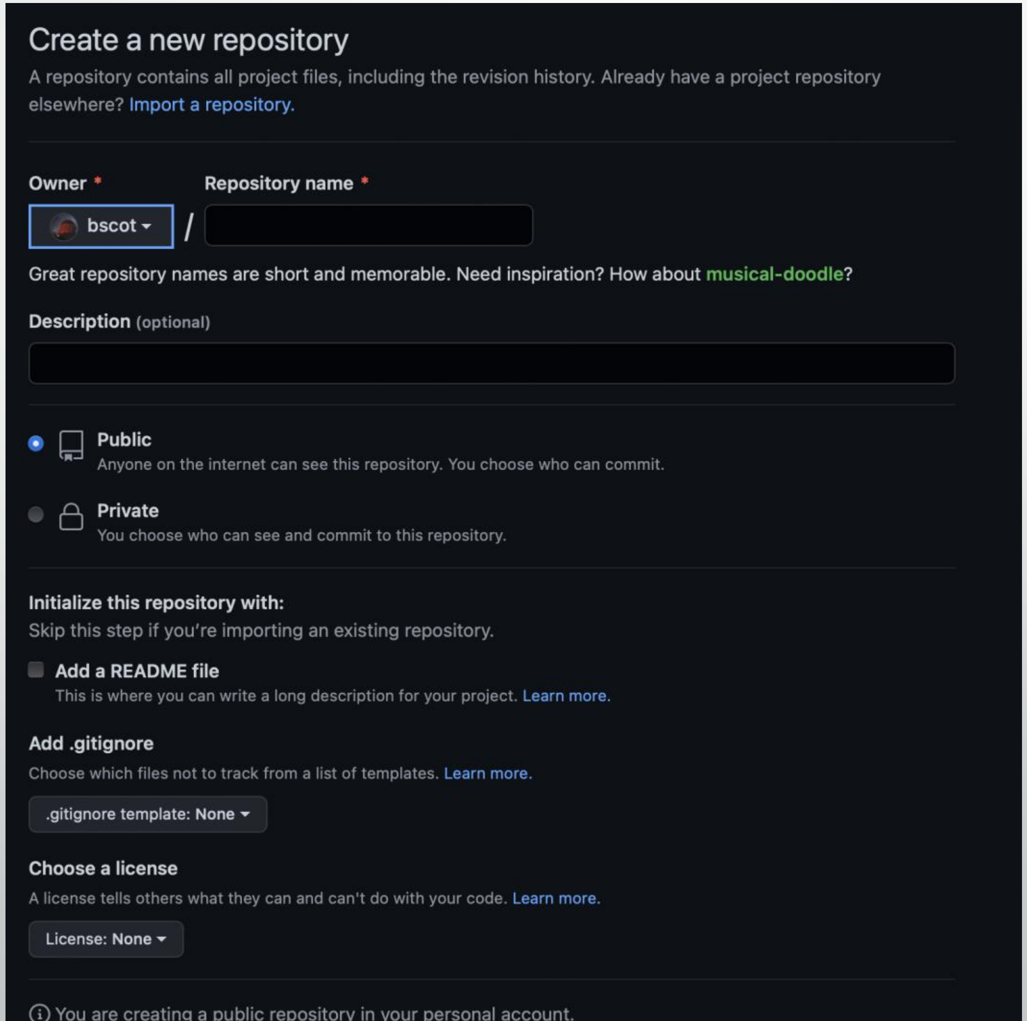
Introduction to Github

+ branches, merges, and more

Git as a distributed version control system

- Remember, git is a distributed version control system. Up until now, we have considered only a local copy of the software repository on our machine.
- But there's nothing special about our copy – we can have n copies of our project that are shared across many developers. We'll need to figure out how to manage n copies (not easy!), but the power of being able to share and collaborate is obvious!
- It would help if we had a convenient place to store a copy that we'll all work from – this is where **github** or **gitlab** come in.

Create a remote repository on github



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner * bscot / Repository name *

Great repository names are short and memorable. Need inspiration? How about musical-doodle?

Description (optional)

● Public
Anyone on the internet can see this repository. You choose who can commit.

○ Private
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

□ Add a README file
This is where you can write a long description for your project. Learn more.

Add .gitignore
Choose which files not to track from a list of templates. Learn more.

.gitignore template: None

Choose a license
A license tells others what they can and can't do with your code. Learn more.

License: None

You are creating a public repository in your personal account.

Linking the local and remote repositories

- Once we have a remote repository, we can link the local and remote respos with git *remote* add [name] [link]. By convention, we use origin as the name for the remote.

```
Git remote add origin [link to your github repo]
```

- We can then *push* the current commit to the remote repo with the famous command,

```
git push origin main
```

- Where origin is the [remote name] and [main] is the branch we want to push.

Caution: the command git push origin *master* was the default until recently. Main has replaced it as the preferred/default name for your production branch.

IN CASE OF 🔥
git commit
git push
git out of building

When you're dead but remember you forgot to git commit git push your last code iterations
gettyimages
gettyimages

Git branches

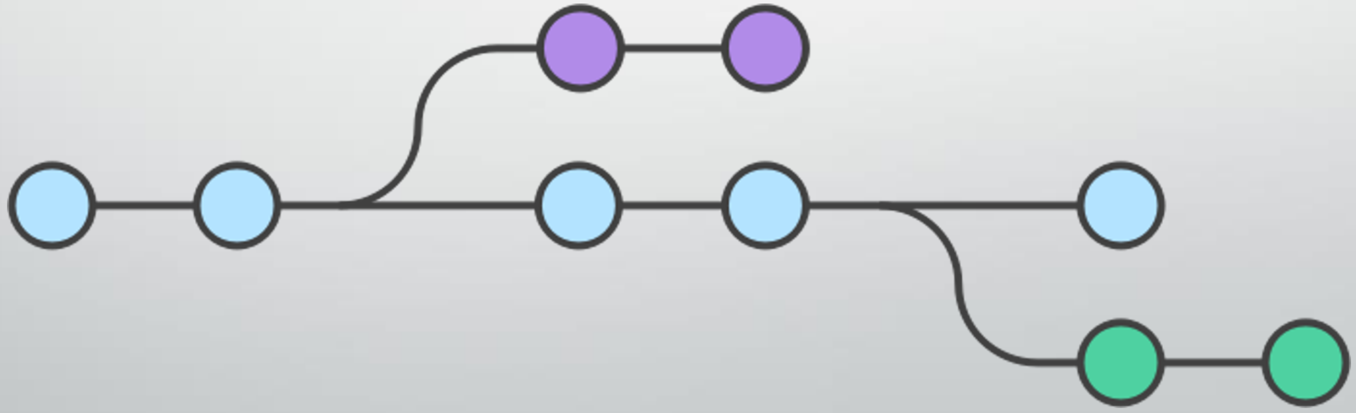
- Let's say we want to add a new feature to our code. We don't want to impact our previous version with bugs that the new feature introduces.

- We could copy the project and work on a whole new copy, perhaps, project_revised.py would be a new file in the copy. Or we could branch the project – recognizing we're still working on the same project, just adding something to it.

To branch our project

- The branch command is

```
git branch [branch_name]
```



Giving branch the –d flag will delete the referenced branch

What is a branch, "really"?

- What git stores is a snapshot of the project with references to unchanged files in previous commits, and copies of changed files in the current commit.

- A branch is just a reference to a specific commit – the *parent.* Each commit has at least one parent. As you make commits, the reference to the parent moves along the branch. The HEAD variable points to the name of the branch we are currently on. Any commits you make will have the HEAD reference as the parent.

Switching to the branch

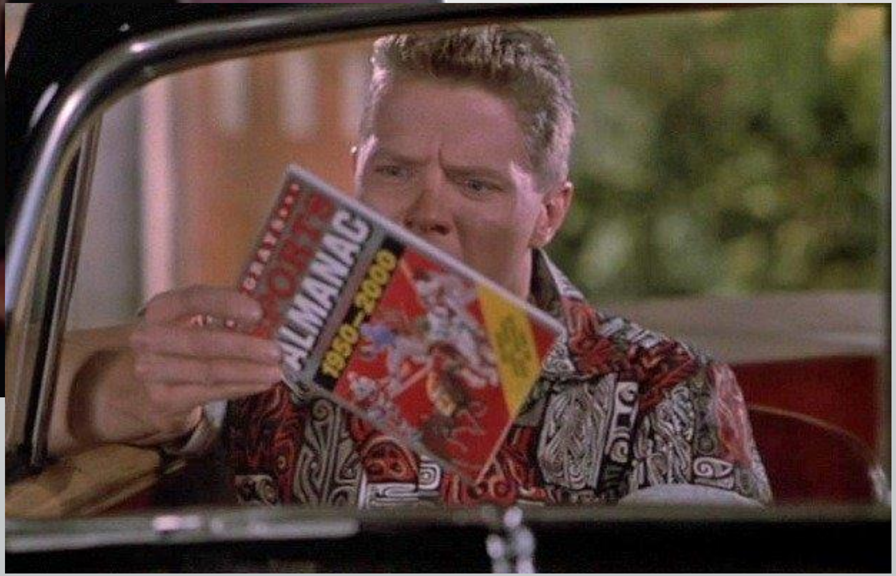
- Now that we've created a branch, we need to switch to it in order to add our new feature. We change branches with the git *checkout* command.

```
git checkout develop
```

- This is very powerful – it allows us to make changes to parts of the project without effecting other parts, or to revert those changes if bugs were introduced.

Git checkout:

BACK TO THE FUTURE



Merges

- We can *merge* two branches together. Merging reproduces all of the commits on one branch in another.

```
git merge [name_of_branch_to_be_merged]
```

- If we are on the branch that we want to merge changes into, we do this with,

```
git push origin [name_of_branch_to_be_merged]
```

- We can also push the branch to the remote with

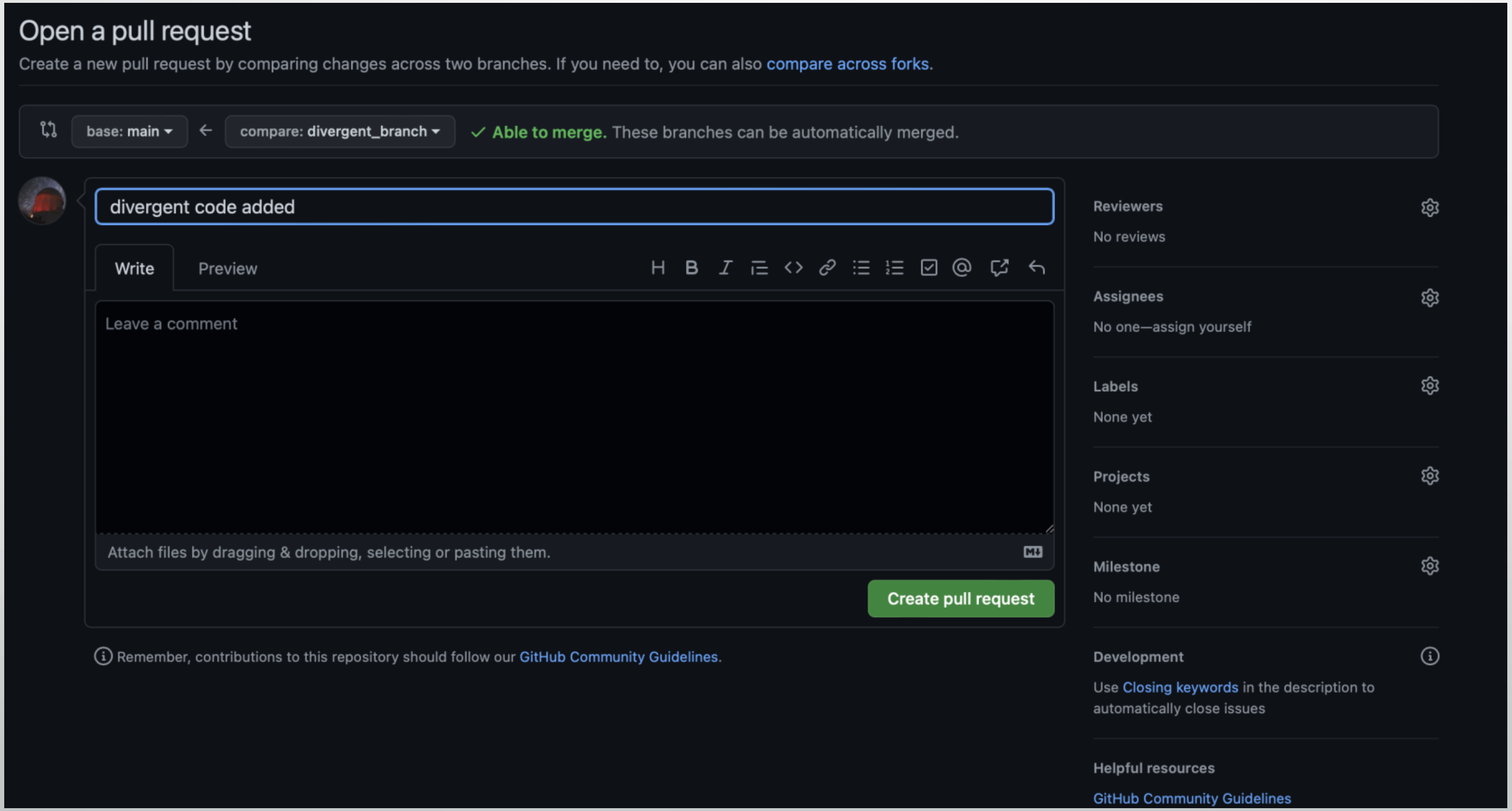
```
git push origin [name_of_branch_to_be_pushed]
```

Pushing branches to remote – Pull Requests

- The output of git pushing a branch to the remote repository will either perform a merge automatically or create a pull request for you to handle merge conflicts manually.
- If there are conflicts, you can go to the github repo and create the pull request through the github GUI. It is also possible to perform this locally. This will merge the branch into the remote repo.

The terminology here is a little weird and confused me for a long time. Push and pull are just opposite actions based on my perspective – am I copying into my branch or copying out of my branch. Remember: git is distributed, so relationships between repos are symmetric.

Github pull requests



Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks.

base: main ← compare: divergent_branch ✓ Able to merge. These branches can be automatically merged.

divergent code added

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Remember, contributions to this repository should follow our GitHub Community Guidelines.

Reviewers
No reviews

Assignees
No one—assign yourself

Labels
None yet

Projects
None yet

Milestone
No milestone

Development
Use Closing keywords in the description to automatically close issues

Helpful resources
GitHub Community Guidelines

Pulling branches from remote

- This works in reverse too. If I want to copy changes in the github repo for my project to my local repo, I use:

```
git pull origin main
```

Or

```
git pull origin
```

Types of merges

- A fast-forward merge brings the local repo up to date with changes made in the remote repo. This kind of merge occurs if the local and remote repos are on the same "timeline", that is, if the remote repo is the child of the local repo.

- A 3-way merge occurs when there is no simple parent child relationship that can be used to generate the merge. This happens if we change the same file across different branches. Git handles this with a two-step commit.

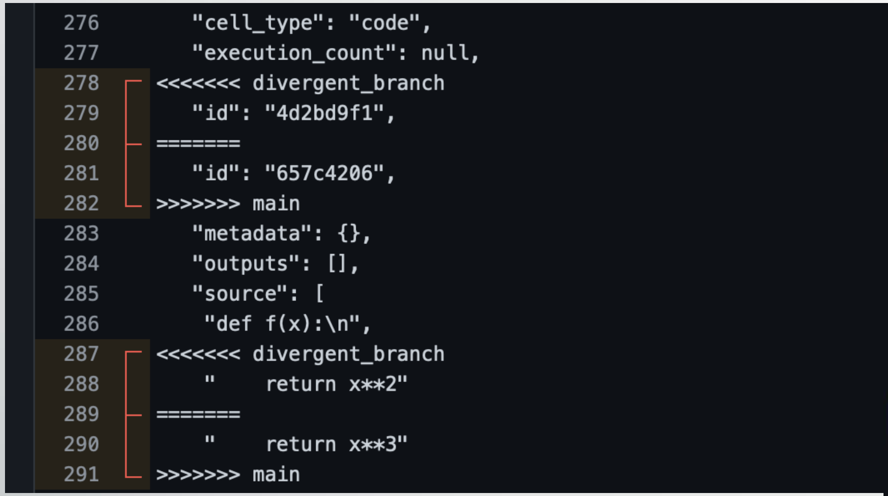
Merge Conflicts

- In a 3 way merge, git first performs a *fetch* action where the branch to be merged is copied into the repository we're working in. A git reference called FETCH_HEAD is created that points to this copy.

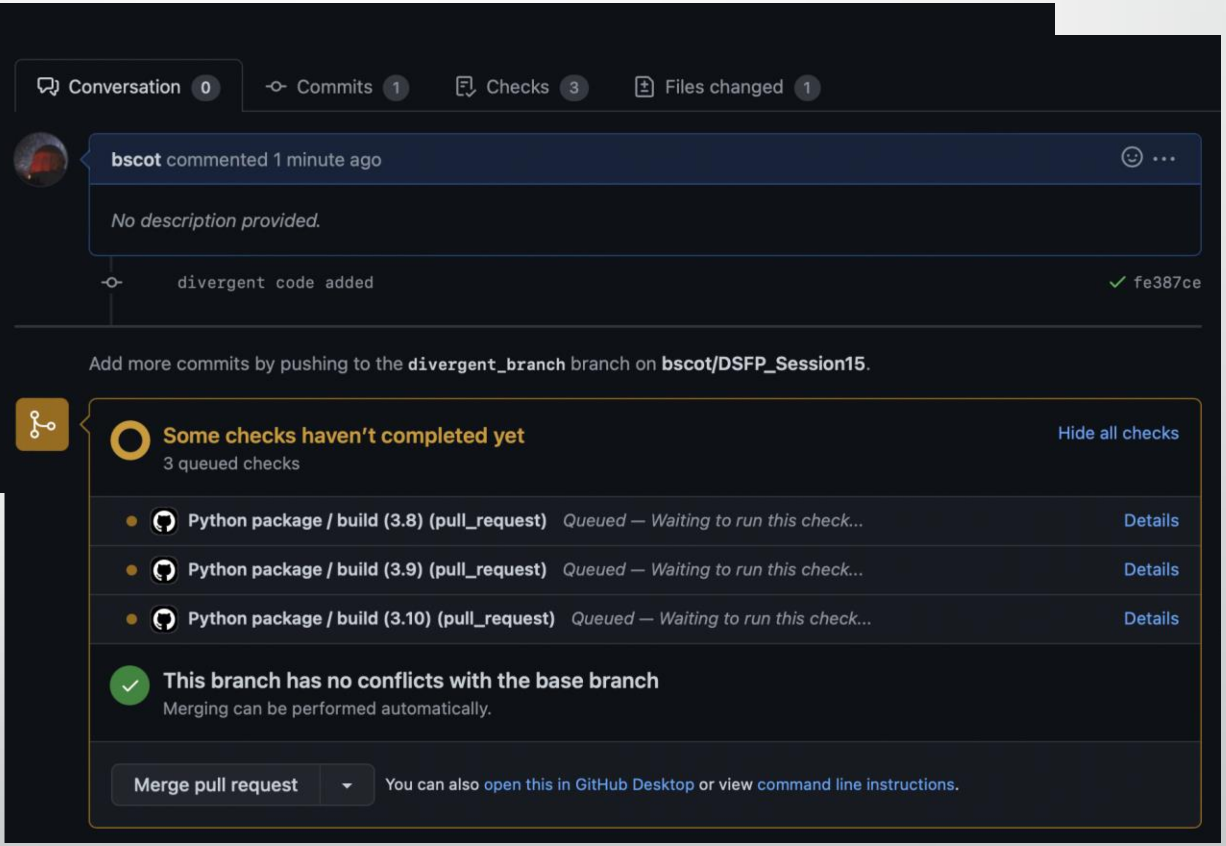
- Git then attempts to make the merge, which produces a conflict. The repository is now in the merging state, where we need to resolve the conflicts to complete the merge.

- We manually fix the conflicts and commit the changes.

Merge Conflicts



```
    "cell_type": "code",
    "execution_count": null,
<<<<<<< divergent_branch
    "id": "4d2bd9f1",
=======
    "id": "657c4206",
>>>>>>> main
    "metadata": {},
    "outputs": [],
    "source": [
     "def f(x):\n",
<<<<<<< divergent_branch
     "    return x**2"
=======
     "    return x**3"
>>>>>>> main
```



bscot commented 1 minute ago

No description provided.

divergent code added ✓ fe387ce

Add more commits by pushing to the **divergent_branch** branch on **bscot/DSFP_Session15**.

Some checks haven't completed yet
3 queued checks

- **Python package / build (3.8) (pull_request)** *Queued — Waiting to run this check...* Details
- **Python package / build (3.9) (pull_request)** *Queued — Waiting to run this check...* Details
- **Python package / build (3.10) (pull_request)** *Queued — Waiting to run this check...* Details

This branch has no conflicts with the base branch
Merging can be performed automatically.

You can also open this in GitHub Desktop or view command line instructions.

Putting this into practice

In the DSFP repository, you will find a notebook that walks you through an example git workflow with data from SDSS.

ME WHEN I SUBMIT A PULL REQUEST
ME WHEN I APPROVE MY OWN PULL REQUEST
imgflip.com

In tomorrow's session, we'll use this example to demo continuous integration workflows.

Introduction to Continuous Integration

With unit tests and github actions

Learning Outcomes

By the end of this talk, you should be able to *automate* the *unit testing* of your code.

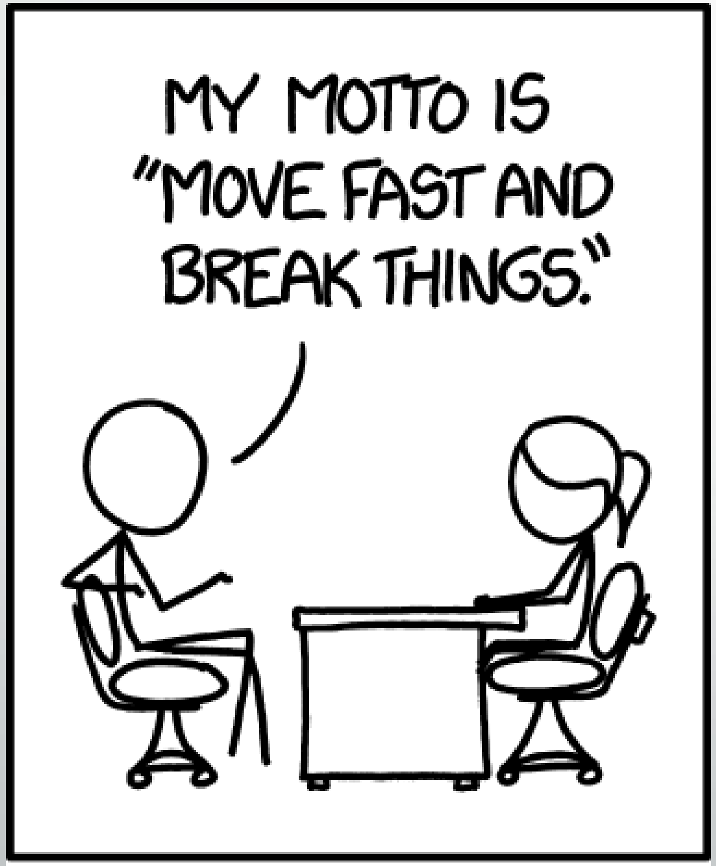
You should also understand how to do *test driven* development.

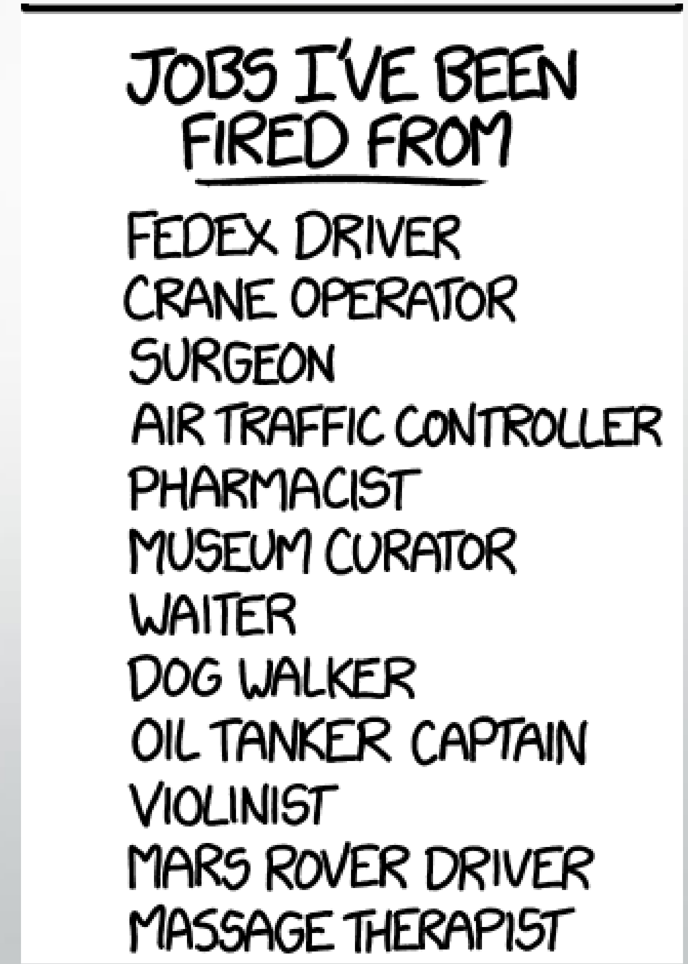
Literature this talk is based on: C. Chandrasekara, P. Herath *Hands-on GitHub Actions* and DSFP Session 3.

How do you manage a large project?

- As we saw yesterday, distributed version control provides a large number of benefits for managing software and ensuring scientific reproducibility
- Historically, software projects were treated like other engineering tasks, a large amount of planning and careful checks prior to *production*.
- This model has largely been replaced with the concept of *agile development.*

"Move fast and break things."



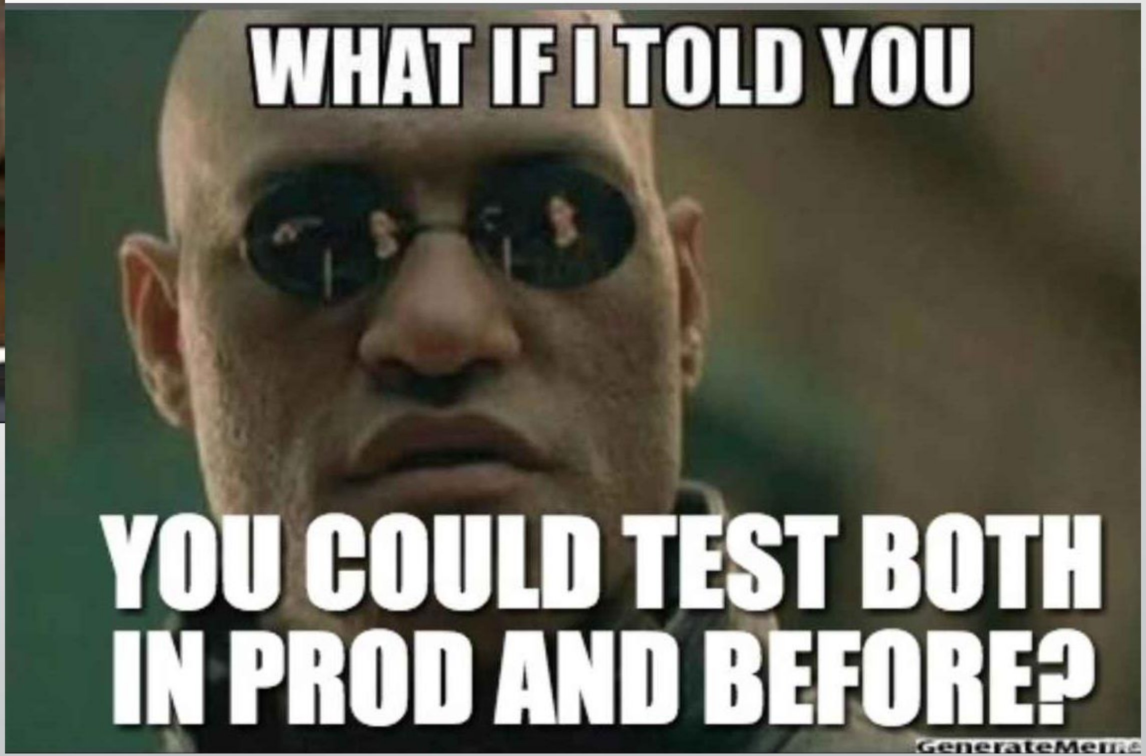


The **big idea** is to build code around testing, rather than planning. Write a test for a method, implement the method, then apply the test.

Continuous Integration

Continuous integration is a development paradigm in which changes to a software project are made and tested often. Usually these tests are automated and distributed.

Scientists Test on Production



Software testing paradigms

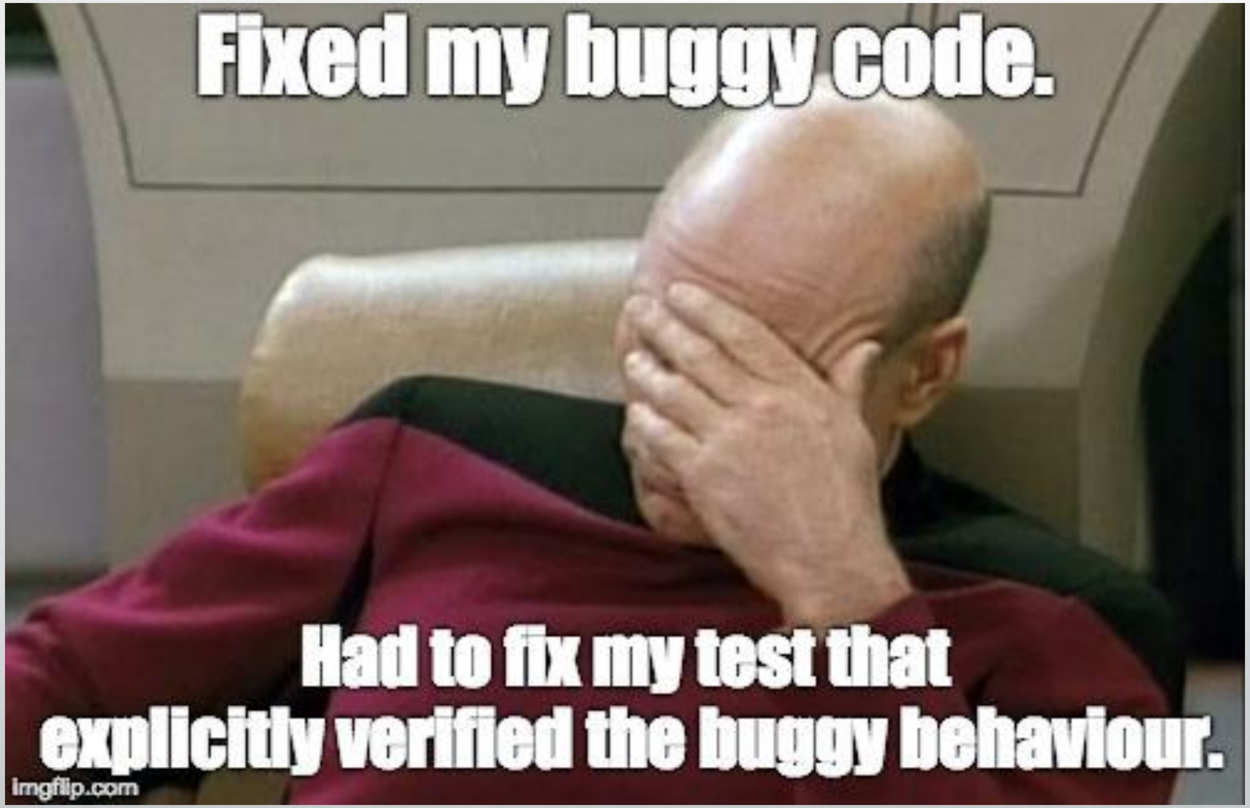
- Scientific workflows often involve *closed box* or *integration tests*. We write code and confirm that it reproduces specific results from the literature. We then trust it to give us reliable results when applied to new data, independent of the internal code mechanisms.

- We are less used to 'open box' unit tests – where individual parts of our code are tested early and often.

Principles of good unit tests

- Independence/modularity
- Simple and Fast
- Deterministic

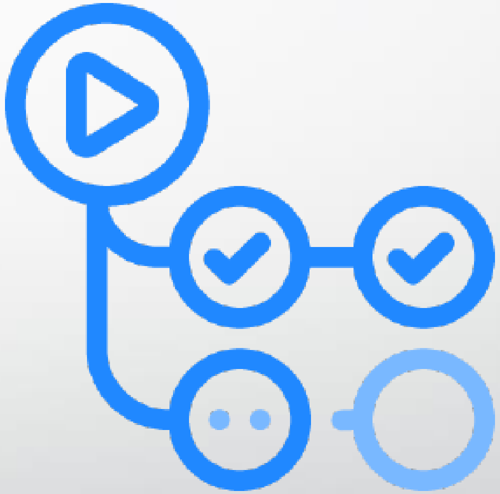
Unit tests can introduce bugs too



Example Unit Test

```
def add(a,b):
    c = a + b
    return c


def test_add(a, b):
    Added = add(1,1)
    assert added = 2
```

Pytest, Continuous Integration, and Github Actions



Pytest directory structure

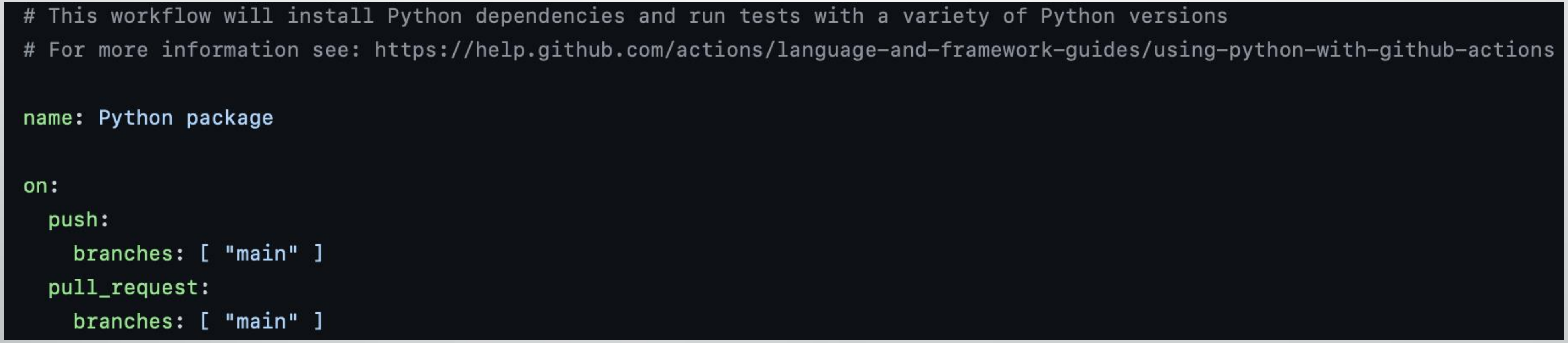
- Pytest is fairly inflexible, it expects a certain directory structure:
  - Packagename/__init__.py
  - Packagename/module_name.py
  - Packagename/test_module_name.py
    - Packagename/tests/test_module_name.py

Checking Coverage

- To have confidence in your code, tests should *cover* as much of your code as possible.
- We can check coverage with the pytest –coverage option.

Configuring Github Actions

- Github actions uses YAML - "YAML Ain't Markup Language" configuration files. These have a specific key: value pair syntax. For example,



```yaml
# This workflow will install Python dependencies and run tests with a variety of Python versions
# For more information see: https://help.github.com/actions/language-and-framework-guides/using-python-with-github-actions

name: Python package

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
```

(More) on Configuring github Actions

Key: value pairs in a nested structure, so for, example, to setup tests to run automatically on a push to the main branch

```
on:
    push:
        branches: ["main"]
```

Can also schedule to run at a certain time every day with

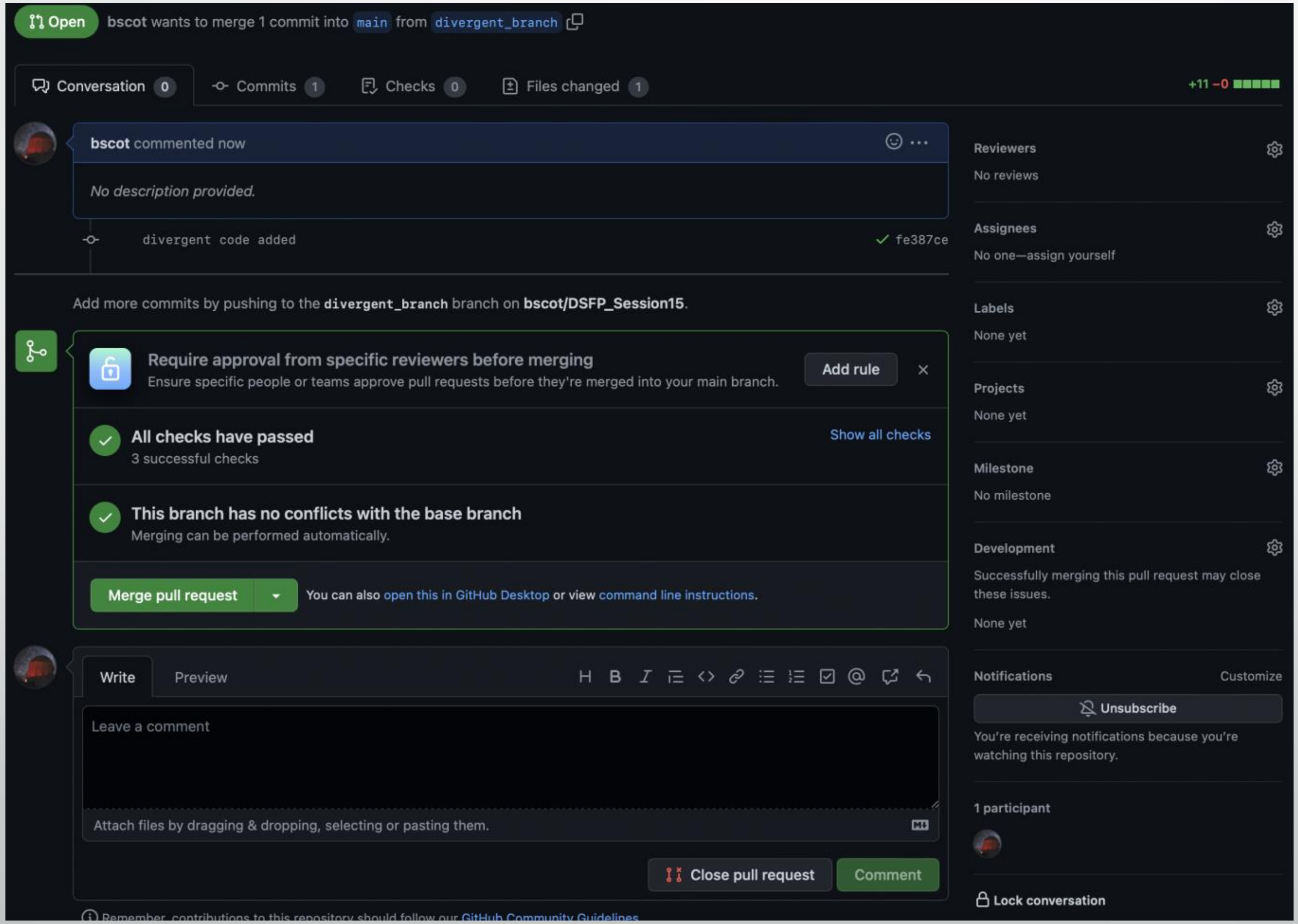
```
on:
    schedule:
        chron: * 12 * * *
```

Configuring Github Actions: Jobs

Jobs are the "content" of a github action – they are the collection of all of the things that you want to occur. The syntax is:

```
Jobs:
     Build:
        runs-on: <virtual machine> #"runner"
      Steps:
        - name: Install dependencies
          run:
         # (some code to install dependencies)
       - name: run tests
          run:
         # (some code to run tests)
```

Continuous Integration in Github Pull Requests



bscot wants to merge 1 commit into `main` from `divergent_branch`

bscot commented now

No description provided.

`divergent code added` ✓ `fe387ce`

Add more commits by pushing to the **divergent_branch** branch on **bscot/DSFP_Session15**.

Require approval from specific reviewers before merging
Ensure specific people or teams approve pull requests before they're merged into your main branch.

✓ **All checks have passed**
3 successful checks

✓ **This branch has no conflicts with the base branch**
Merging can be performed automatically.

You can also open this in GitHub Desktop or view command line instructions.

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Remember, contributions to this repository should follow our GitHub Community Guidelines.

Reviewers: No reviews

Assignees: No one—assign yourself

Labels: None yet

Projects: None yet

Milestone: No milestone

Development: Successfully merging this pull request may close these issues.
None yet

Notifications: You're receiving notifications because you're watching this repository.

Unit Test Exercise

- In the DSFP Session 15 Repository, you'll find a notebook that will walk you through setting up a github actions workflow and writing some unit-tests. You'll also find a template .yml file.
- We will continue to work on the clustering example from yesterday. You should have some cluster centers to work with, but if you do not, you can generate some mock cluster centers.