12.010 Computational Methods of Scientific Programming 2021

Lecture 22: Version control and software management practices

Summary

- Version control and related tools, motivation
- Using “Git” and “Github”
  - status, log, add, commit, fork, clone, push, remote, branch, pull request….
- Adding automated testing
  - CI/CD
- Publishing
  - Zenodo
  - JOSS
  - Containers

Towards FAIR principles for research software

http://dx.doi.org/10.3233/DS-190026

Note –
Some specific tools barely existed 5 years ago! They are useful, but the tools here are an area of frequent development.
The concepts are more durable.

Version control

- In software projects it is common for code to evolve through multiple "versions" this can result in potential for confusion. Programs may unexpectedly stop working when functions/data structures that depend on each other get out of sync etc…

- Version control is used to
  - keeping track of versions of files
  - avoid confusion over what bits of code are being used to build a program or system

Versions of python packages on 12.010 cloud system.

```
cnh@mit.edu@ip-172-30-1-201:~$ conda list -e | head -50
# This file may be used to create an environment using:
# $ conda create --name <env> --file <this file>
# platform: linux-64
_libgcc_mutex=0.1=conda_forge
_openmp_mutex=4.5=1_gnu
_r-mutex=1.0.0=anacondar_1
affine=2.3.0=pypi_0
aiohttp=3.7.4.post0=py37h7f8727e_2
alembic=1.6.5=pypi_0
alsa-lib=1.2.3=h516909a_0
antlr-python-runtime=4.7.2=py37h89c1867_1002
anyio=3.3.0=pypi_0
appdirs=1.4.4=pyhd3eb1b0_0
argcomplete=1.12.3=pyhd3eb1b0_0
argon2-cffi=20.1.0=py37h27cfd23_1
asciitree=0.3.3=py_2
asdf=2.8.1=pyhd8ed1ab_0
astropy=4.3.1=py37h09021b7_0
async-timeout=3.0.1=py37h06a4308_0
async_generator=1.10=py37h28b3542_0
atk-1.0=2.36.0=h28cd5cc_0
attrs=21.2.0=pyhd3eb1b0_0
awscli=1.20.58=py37h89c1867_0
babel=2.9.1=pypi_0
```

```
find: './.cph_tmp7h2d7717': Permission denied
  63269   63269 5253480
```

About 450 packages in total, each with their own version.

Each package is made up of multiple python files (>60,000)

Keeping things like this all “consistent” is the origin of version control and related tools

For software used in research these sorts of tools are important for reproducibility, sharing, collaboration etc…

Modern version control/software reproducibility “tooling”

- Git
  - Git has become the standard tool for version control
  - Free online services like “Github” and “Gitlab” provide a nice web interface and make Git easier to use
  - Git itself is a software tool separate from Github/Gitlab. Github/Gitlab make collaborating using git easier.

- Continuous Integration/Continuous Deployment (CI/CD)
  - Fancy name for automated testing to avoid breaking things by mistake
  - Online services like Github provide ways to integrate CI/CD into version control environment

- Publishing
  - Zenodo/JOSS – DOI to reference/cite/find software versions
  - Containers - A system for capturing specification of a whole computer + application to help make things portable.

Git basics

- The best way to understand git is to use it, following some tutorial. It makes a lot more sense in use, than on paper!
- Git organizes collections of code in “repositories”
- A repository is just a directory tree with files and some special information in a “.git” subdirectory

Creating a new repository by hand

```
$ mkdir my_cool_software_repository
$ cd my_cool_software_repository
$ git init
Initialized empty Git repository in /home/jpy_class/mit/12.010/cnh@r
tory/.git/
$ ls -altr
total 20
drwxr-x--- 29 cnh@mit.edu cnh@mit.edu 12288 Dec  2 01:31 ..
drwxr-xr-x  3 cnh@mit.edu cnh@mit.edu  4096 Dec  2 01:31 .
drwxr-xr-x  7 cnh@mit.edu cnh@mit.edu  4096 Dec  2 01:31 .git
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
$
```

A git repository has a special subdirectory ".git" at its "root".
Files and directories can be added.
.git keeps track of history of "committed" adds/deletes and edits.

Add a file

```
$ echo 'print("hello")' > hello.py
$ python hello.py
hello
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be c
        hello.py

nothing added to commit but untracked files present (us
$ git add hello.py
$ git status
On branch master

No commits yet

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

$
```

Files and directories can be created in the repository directory tree.

Note:

Don’t do anything in .git

Git is designed for files of code and text files. It does not work well with images, binary files etc…

Here we created a file “hello.py” and added it to the current, active files tracked by git.

To make the change a permanent we must “commit” changed file.

Commit the changes

```
$ git commit -m "add a first file" hello.py
[master (root-commit) 195aba7] add a first file
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
$ git status
On branch master
nothing to commit, working tree clean
$ git log
commit 195aba73130b4b5f701cebffa85e90233f40dde0 (HE
Author: Chris Hill <cnh@mit.edu>
Date:   Thu Dec 2 01:42:16 2021 +0000

    add a first file
$
```

```
$ git show
commit 195aba73130b4b5f701cebffa85e90233f40dde0
Author: Chris Hill <cnh@mit.edu>
Date:   Thu Dec 2 01:42:16 2021 +0000

    add a first file

diff --git a/hello.py b/hello.py
new file mode 100644
index 0000000..11b15b1
--- /dev/null
+++ b/hello.py
@@ -0,0 +1 @@
+print("hello")
$
```

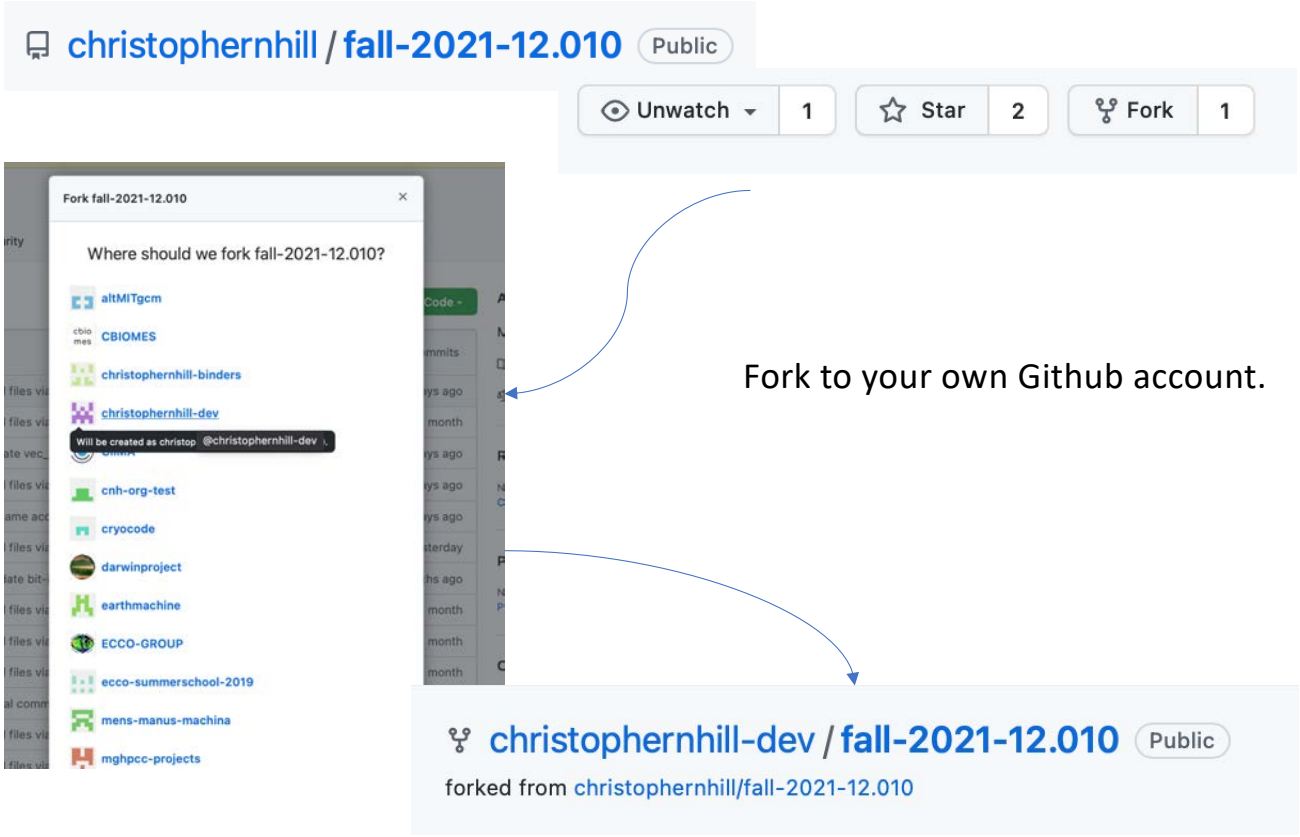
In git, in addition to adding any new files, you ”commit” changes that you want to keep.

The sequence of commits and changes is stored in the .git directory so that you can compare different versions of files, directories of repositories.

The commit id number is unique, so you can compare any commit with another.

Working with Git via Github

- Git on its own can be a bit fiddly to use, especially for sharing and publishing code and remote collaboration. Lots of projects use Github to streamline workflow.
- Creating a copy of an existing repository, making a change and then submitting change back to original repository is a good way to start. In Github this starts with a “fork” of the existing repository.



After creating a “fork”, download to local compute using “clone”.

```
$ git clone https://github.com/christophernhill-dev/fall-2021-12.010.git
Cloning into 'fall-2021-12.010'...
remote: Enumerating objects: 418, done.
remote: Counting objects: 100% (418/418), done.
remote: Compressing objects: 100% (390/390), done.
remote: Total 418 (delta 182), reused 24 (delta 9), pack-reused 0
Receiving objects: 100% (418/418), 3.72 MiB | 9.35 MiB/s, done.
Resolving deltas: 100% (182/182), done.
$ cd fall-2021-12.010
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
$
```

Now lets make a change – we do this on a "branch"

```
cnh@mit.edu@ip-172-30-1-201:~/fall-2021-12.010$ git checkout -b chris/small-test-edit
Switched to a new branch 'chris/small-test-edit'
cnh@mit.edu@ip-172-30-1-201:~/fall-2021-12.010$ git status
On branch chris/small-test-edit
nothing to commit, working tree clean
cnh@mit.edu@ip-172-30-1-201:~/fall-2021-12.010$ echo 'print("hello")' > hello.py
cnh@mit.edu@ip-172-30-1-201:~/fall-2021-12.010$ git status
On branch chris/small-test-edit
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        hello.py

nothing added to commit but untracked files present (use "git add" to track)
cnh@mit.edu@ip-172-30-1-201:~/fall-2021-12.010$

cnh@mit.edu@ip-172-30-1-201:~/fall-2021-12.010$ git add hello.py
cnh@mit.edu@ip-172-30-1-201:~/fall-2021-12.010$ git commit -m "add hello" hello.py
[chris/small-test-edit 47d85bf] add hello
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
cnh@mit.edu@ip-172-30-1-201:~/fall-2021-12.010$ git status
On branch chris/small-test-edit
nothing to commit, working tree clean
cnh@mit.edu@ip-172-30-1-201:~/fall-2021-12.010$
```

There are lots of ways to use Git.

A common practice is to create a "branch" for a set of edits.

The default branch is called master or main, here we create a branch with a name to remind us what it is for.

Once we have created our file we need to add and commit.

This records our new changes locally, associated with the branch.

After we are happy with our local changes we can “push” them to our repository fork.

```
cnh@mit.edu@ip-172-30-1-201:~/fall-2021-12.010$ git push
fatal: The current branch chris/small-test-edit has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin chris/small-test-edit

cnh@mit.edu@ip-172-30-1-201:~/fall-2021-12.010$ git push --set-upstream origin chris/small-test-ed
it
Username for 'https://github.com': christophernhill
Password for 'https://christophernhill@github.com':
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 2 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 283 bytes | 283.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'chris/small-test-edit' on GitHub by visiting:
remote:      https://github.com/christophernhill-dev/fall-2021-12.010/pull/new/chris/small-test-ed
it
```

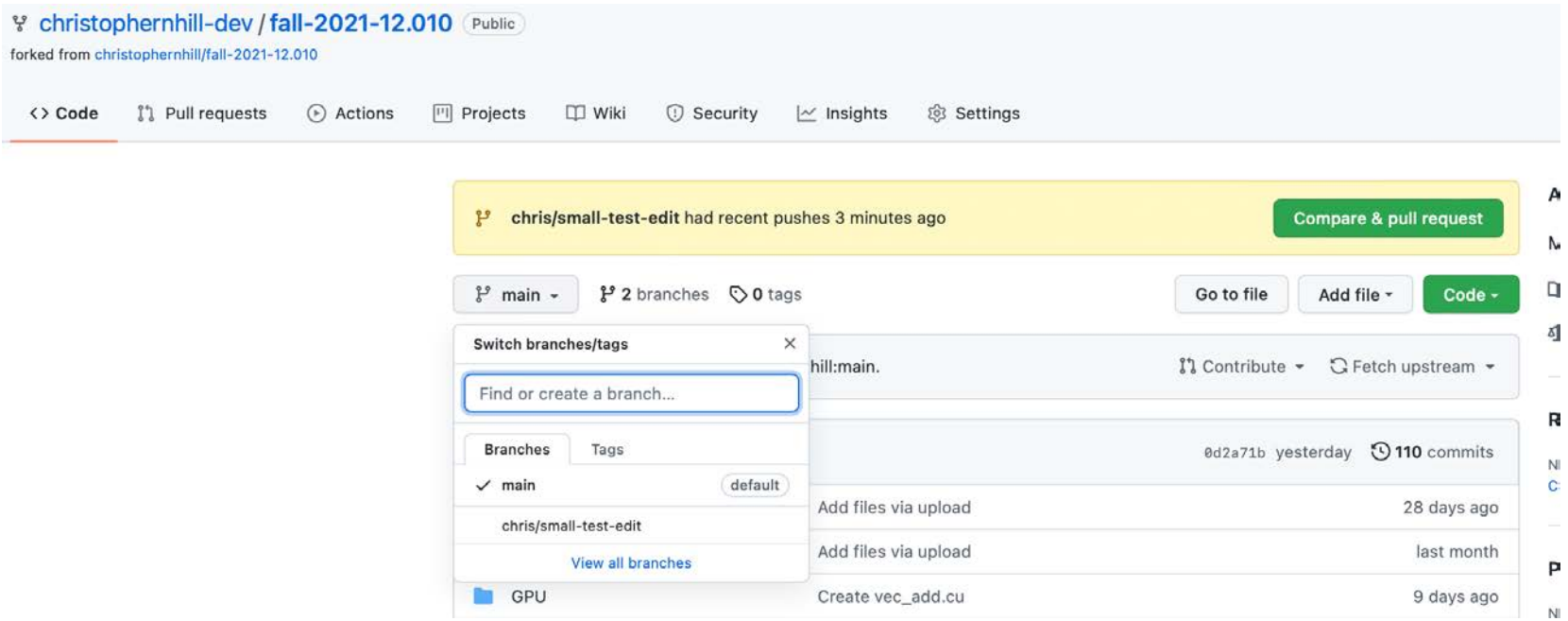
We use “git push” to send changes back to the repository we cloned.

Here we use

git push --set-upstream origin chris/small-test-edit

to “push” changes to our Github repository.

After push our online repo has changes



The online repo also has a “Compare & pull request “ button.

Current state

1. Official repository unchanged
2. Fork of official repository has a “branch” with changes that we made on a local machine and “pushed” to Github.
3. Local machine has clone of our fork. It has branch and main

```
cnh@mit.edu@ip-172-30-1-201:~/fall-2021-12.010$ git diff main
diff --git a/hello.py b/hello.py
new file mode 100644
index 0000000..11b15b1
--- /dev/null
+++ b/hello.py
@@ -0,0 +1 @@
+print("hello")
cnh@mit.edu@ip-172-30-1-201:~/fall-2021-12.010$
```

If we now want to update the official repository we create a ”pull request” (PR).

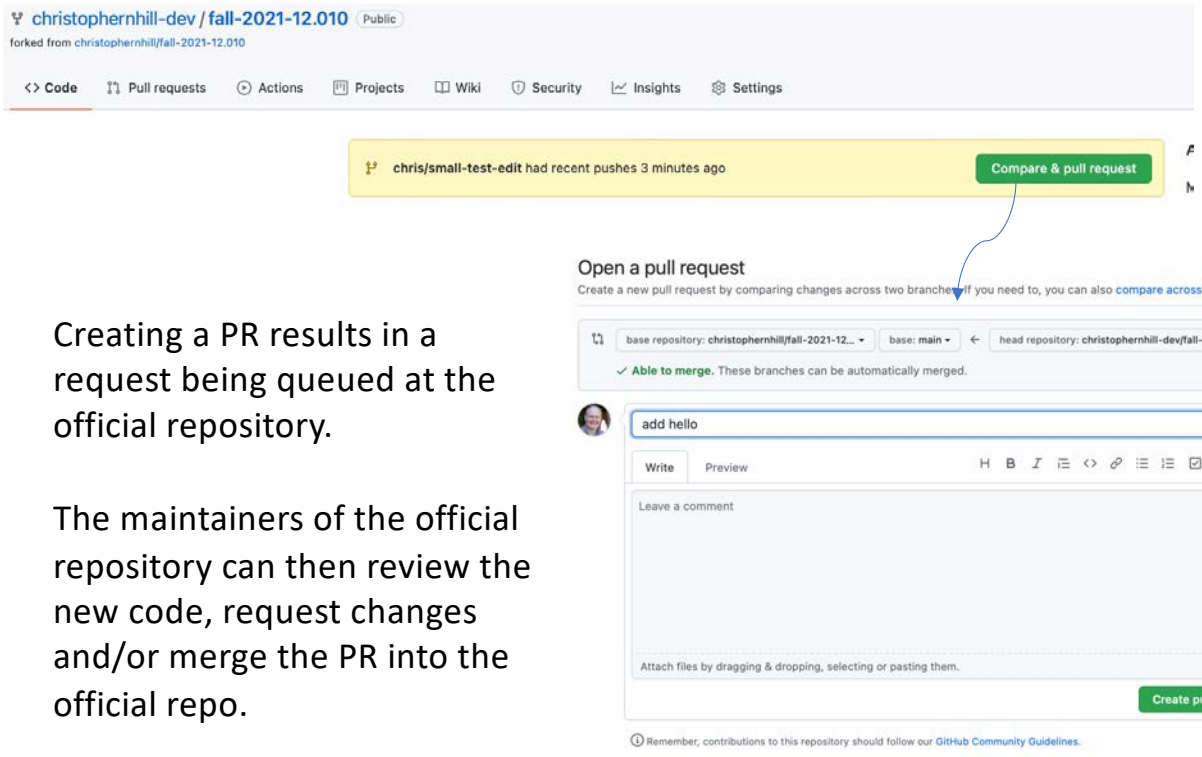
A PR is a way to ask the maintainer of the official repository to incorporate your changes (i.e. pull them into the official repository).

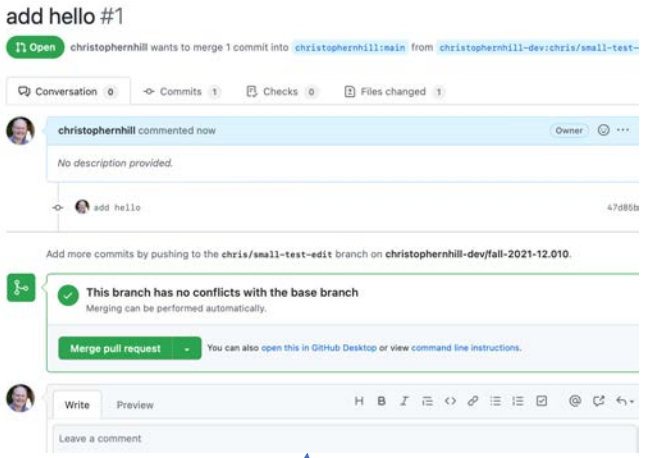
Note –

This workflow can seem somewhat complicated. It is does allow people all over the planet to collaborate on large software.

Git does not know which is the “official” repository. That is a choice of a project. All repositories are peers to Git.

Creating a pull request





Creating a PR results in a request being queued at the official repository.

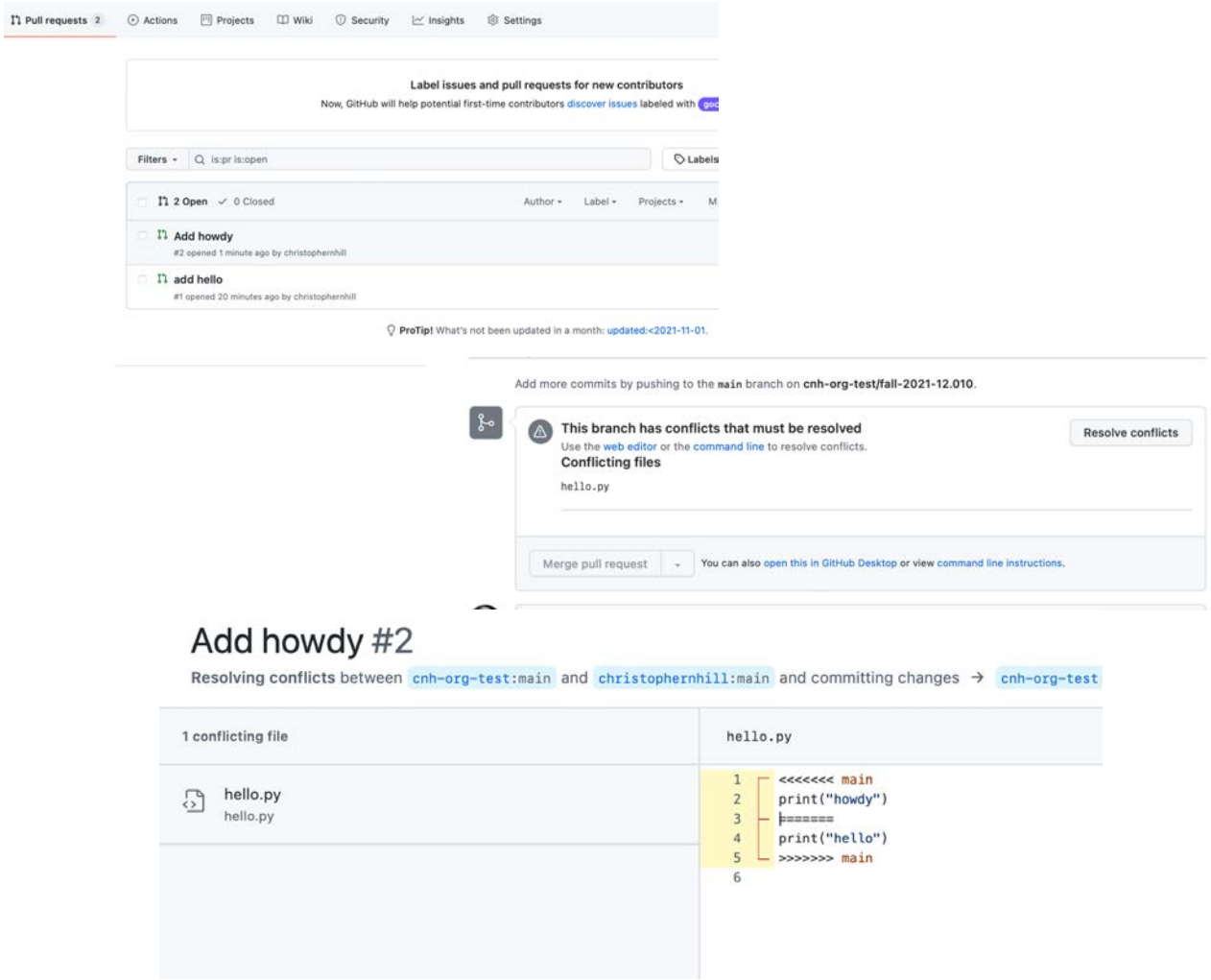
The maintainers of the official repository can then review the new code, request changes and/or merge the PR into the official repo.

Conflicts

Two commits can potentially contain changes to same file.

Git will try and merge if the changes are from the same “base” _and_ they are in different parts of the file.

Otherwise a conflict will be detected and will need to be resolved manually.



Hands on

- Try fork, clone, edit, diff, log, status, commit/add, push, pull request….

Git and Github/Gitlab streamline collaboration and versioning.

- BUT - what about checking if a PR will break something.

- For this testing is important.

- Github/Gitlab have builtin features to help with automated testing.
  - CI/CD – continuous integration/continuous deployment is a name somtimes given to this sort of automated testing. The CI/CD name comes from online software, where the deployment of continually evolving software to web is automated (Facebook, twitter etc….)

Automated testing tools

- These are not part of Git, but Github/Gitlab do provide
  - Github actions is a common tool for automated testing

Automated testing as a concept is an important piece of maintaining software.

This is an area where specific tools are evolving particularly quickly.

We will look at “Github Actions” as an example. It is widely used, but only appeared in 2019.

Automated testing

- General idea is
  - Have some tests that check things still work
    - Whenever new code is added, or changed
    - Regularly (nightly, weekly etc…) in case something else changes that affects code working
  - Automating testing is preferable where possible
  - Tests can be a mix of
    - Unit tests
    - Integration tests
    - System tests
  - Different sets of tests may run at different frequencies
  - Tools like Github, Gitlab have features to help set up these processes

Github actions – automated testing example

- Add a special directory ".github/workflows" to a repository.
- Adds files in that directory that describe "actions" to take in response to events (such as a commit or a pull request)
- The actions are scripts that can perform whatever tests make sense
- Github provides virtual machines that run the scripts when they are triggered.

Github actions – automated testing example

main fall-2021-12.010 / .github / workflows / main.yml

christophernhill Example CI/CD step ✓

1 contributor

34 lines (27 sloc) | 1.05 KB

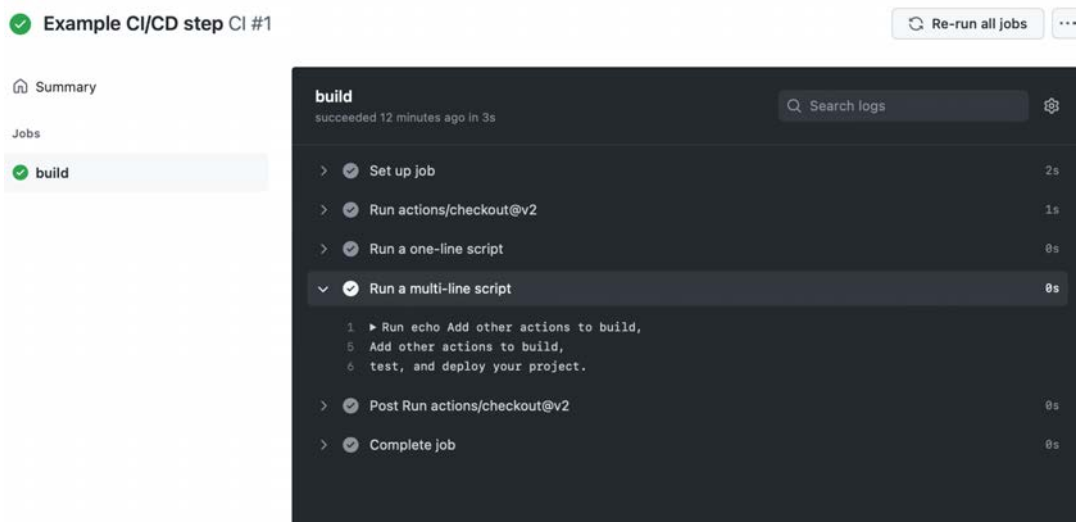
```yaml
# This is a basic workflow to help you get started with Actions

name: CI

# Controls when the workflow will run
on:
  # Triggers the workflow on push or pull request events for any branch
  push:
  pull_request:

  # Allows you to run this workflow manually from the Actions tab
  workflow_dispatch:

# A workflow run is made up of one or more jobs that can run sequentially or in parall
jobs:
  # This workflow contains a single job called "build"
  build:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest

    # Steps represent a sequence of tasks that will be executed as part of the job
    steps:
      # Checks-out your repository under $GITHUB_WORKSPACE, so your job can access it
      - uses: actions/checkout@v2

      # Runs a single command using the runners shell
      - name: Run a one-line script
        run: echo Hello, world!
```

A simple example, provided by default.

It uses Github actions specific syntax (based on a file format called yaml).

Github actions – results

Results of actions can be used to control whether a pull request can be merged.



Hands on

- Try create CI/CD action

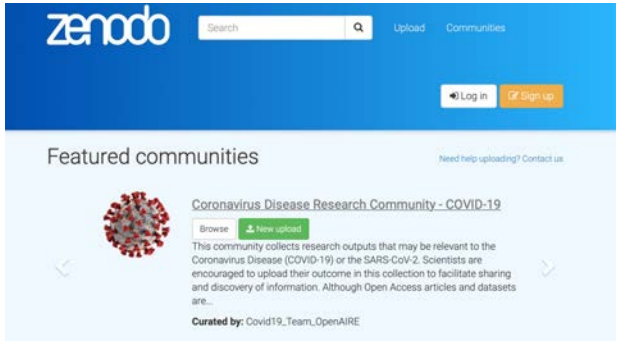
There are other CI/CD systems

- all have a somewhat similar pattern
  - they are controlled by files with specific names/in specific directories
  - they have some high level syntax based on YAML/TOML
  - they can invoke specific commands that are controlled by the repository needs

- Some other examples
  - Travis, buildkite, CircleCI, Gitlab

Publishing software

- Tools like Github provide a new way to “publish” software
- Not quite like a paper sharing science results, but still increasingly useful in research for sharing ideas/techniques.
- There are some useful tools for providing citation, that allow projects using published software to reference properly



This is becoming increasingly common in research communities.

Containers

- In addition to Github there are recent services called "Container registries"

- These can be used with Github to record the entire Operating system and sets of packages used by some software
- See https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1008316

  https://psyarxiv.com/fwxs4/

As software is more and more central to research keeping track of what was used, testing etc… becomes more and more important!

MIT OpenCourseWare

https://ocw.mit.edu

12.010 Computational Methods of Scientific Programming, Fall 2024