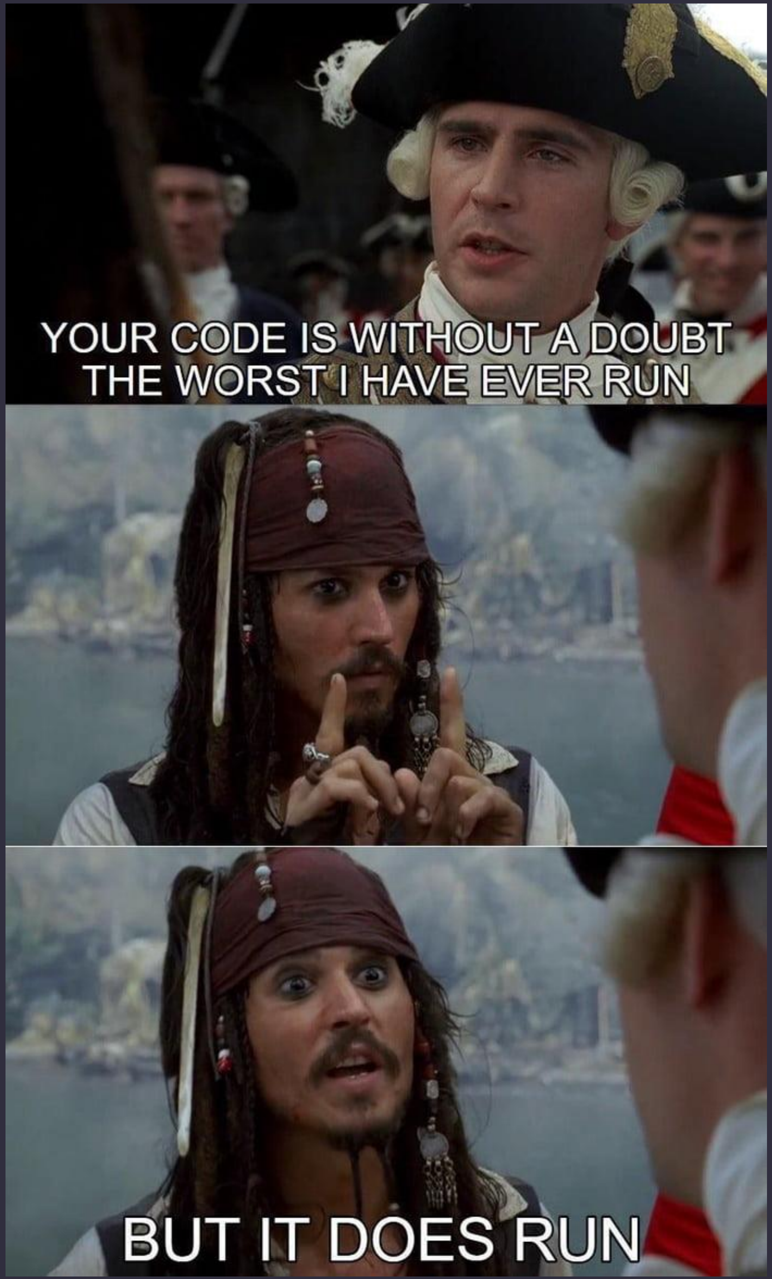
Good Scientific Code

A WORKSHOP ON WHAT, WHY, HOW

BEFORE:



AFTER:

That's got to be the best code I've ever seen.
So it would seem.

Outlook

- Version control with git
- Clear code
- Software developing paradigms
- Collaboration & publishing code
- Documentation
- Scientific project reproducibility

This is a two-days, six-blocks workshop, developed over 3 years, combining textbooks, other workshops, online tutorials from field experts, blog posts, personal experience developing and documenting 10+ software, and research on how to make reproducible science.

Obvious disclaimers:

- These slides describe the ideal scenario
- Writing good code is not instant
- Writing good code is a ***craft***...

1. Iterative process: you constantly improve your skills, and learn more!
2. While general rules guide most of it, details depend on personal opinions!

Housekeeping

- This is a (mostly) language-agnostic workshop, meaning that the principles are about general coding. Examples and exercises will be in Julia and Python
- Workshop is 2 days long:
  - 6 blocks, each 2 hours long, with “lecture” and “exercises” parts intertwined
- Bring your own code base, for a small project (e.g., 2-3 plots of recent paper) which we will hopefully transform from something shameful to something prideful
- You received a link with workshop materials: more will be added at each session
- This workshop has ~120 slides, most with unique information
  - This is baptism by fire. You are not expected to retain everything. You will need to revisit things!

- ***If at any point you wish to ask a question, or discuss what is being presented, please feel free to immediately interrupt and ask away!***

What is the purpose of code?

❑ Technically, code is instructions for the computer…

❑ But, in the end of the day, your code is a mean to solve a problem!

❑ *“Programs are meant to be read by humans and only incidentally for computers to execute.”*, Donald Knuth

❑ Code is written and read by humans! It is just another form of ***technical writing***, similar to a paper: must have ***crystal clear communication*** of the problem solving!

❑ Good Scientific Code =

Clear, Easy to understand, Well-documented, Reproducible, Testable, Reliable, Reusable, Extendable, Generic

Not only you will learn how to achieve all of these in this workshop, but it will be made clear that they aren’t costly.

Choose Your Weapon

- ***Choose what language fits what you want to achieve***
  - Expressivity (how fast and flexibly you can put your ideas to code)
  - An existing library that you want to use
  - Performance is absolutely critical
  - Re-usability and composability with other packages is important
  - I will need to do a lot “science” (plotting, modelling, testing, querying, interactiveness)

- My strong recommendation (because its good for ***all*** of the above bullet points)

julia

Version Control

RETRACEABLE CODE HISTORY USING GIT

The dumb and smart way for code history

- ❑ Dumb: just store a daily backup of your files…
  - Impossible for collaboration
  - Super-duper error prone
  - Large overhead

paper.tex
paper_final.tex
paper_really_final.tex
paper_v1.tex
paper_v2.tex
paper_v2_richard.tex
paper_v3_with_richards_comments.tex

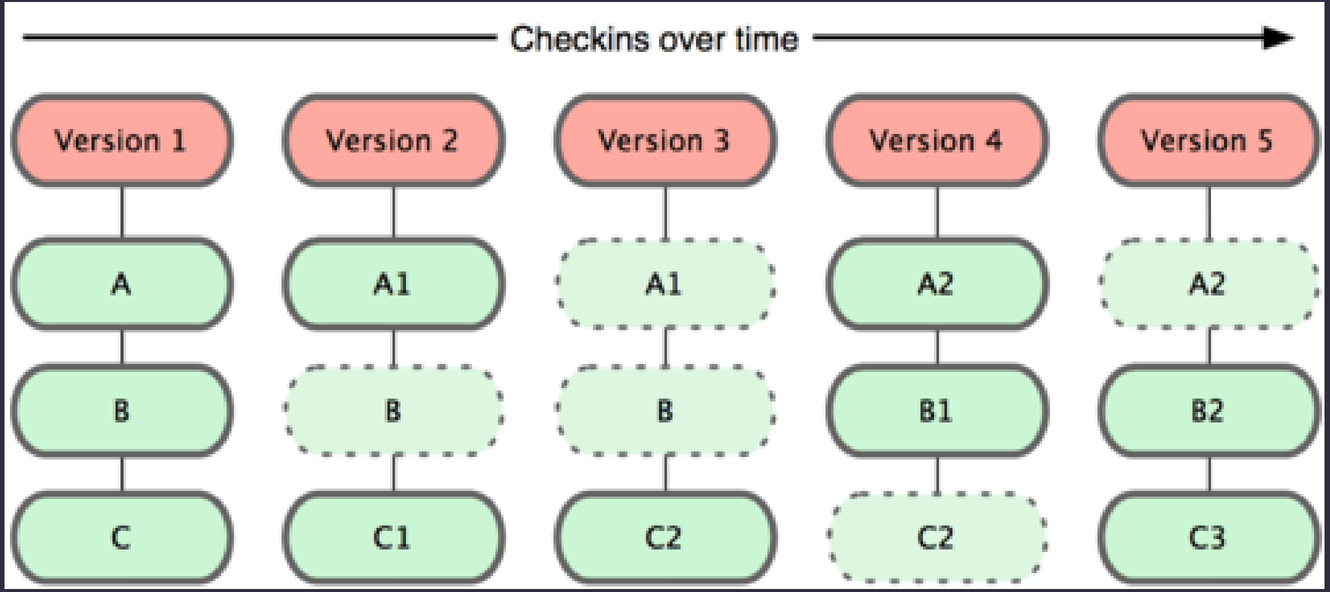
dumb system!

- ❑ Smart: Version Control System
  - Tracks changes in every file in the project ***without*** renaming the file
  - Allows reverting to previous versions of files or entire project without loosing current versions
  - Allows several different instances of your codebase to coexist
  - Facilitates collaborative editing of files by different parties safely
  - Allows tracking who made which changes to the project, hence accessing the chain of blame

- ❑ Because in this course you will be modifying your code base a lot, it’s good to start by learning the system that makes code modification safe and re-traceable

git

- git is a command line tool that is the most used version control system
- It is efficient, because it ***only stores differences*** from previous states



Checkins over time

Version 1	Version 2	Version 3	Version 4	Version 5
A	A1	A1	A2	A2
B	B	B	B1	B2
C	C1	C2	C2	C3

- Any folder can become a git repository via the command `git init`
- Everything is re-traceable, everything can be recovered (even if it is deleted!)
- ***git is the closest thing to time travel humanity has***

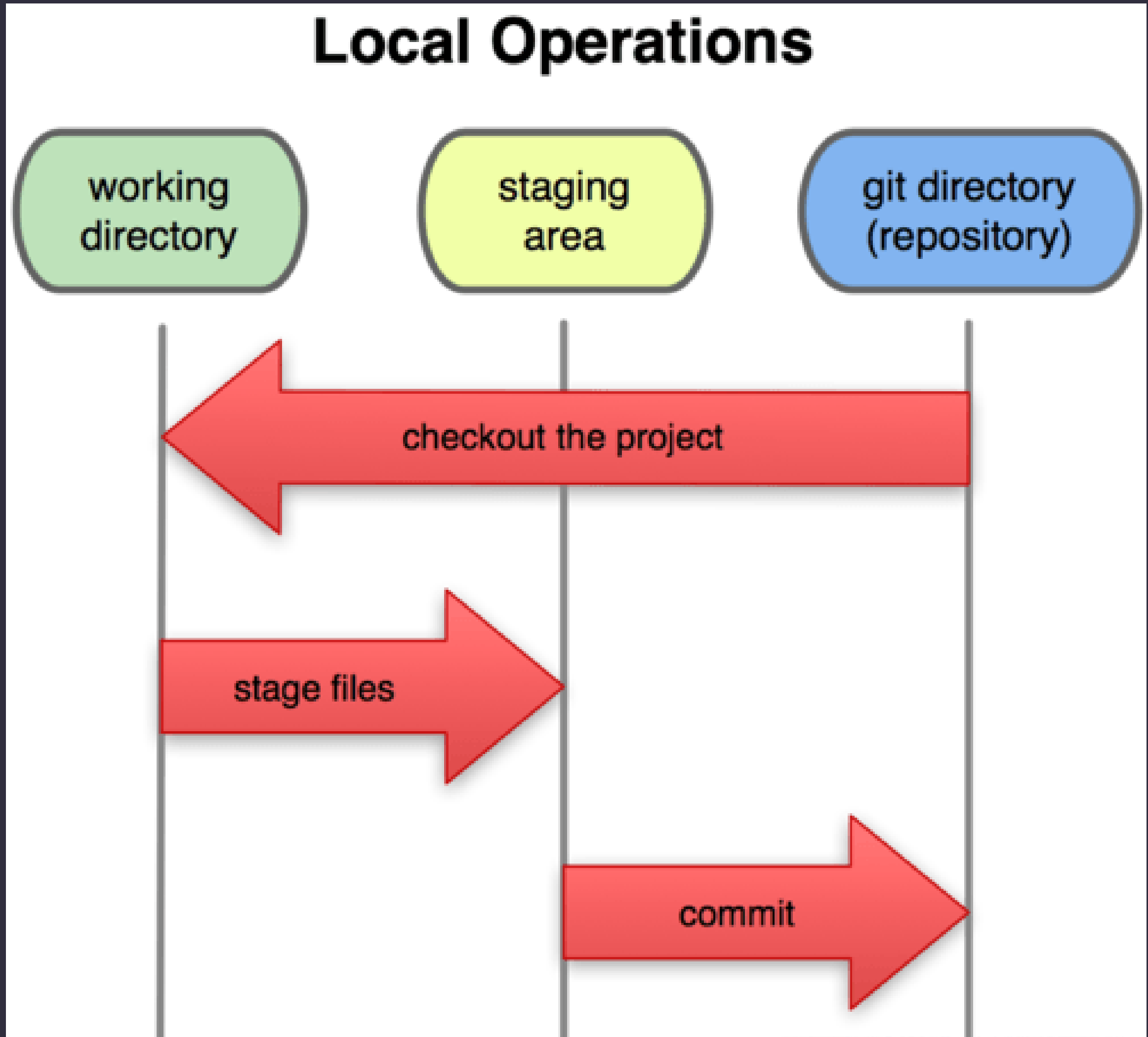
git basics: the three states

- □ Your files can be in three states: modified, staged or committed

- □ Modified means that you have changed the file but have not committed it to your git history yet

- □ Staged means that you have marked a modified file in its current version to go into your next commit snapshot

- □ Committed means that the code is safely stored in the git “history” of your local codebase



git basics: editing, staging, committing

- You edit your files normally, doing a change in your IDE
  - This puts your files in the modified state

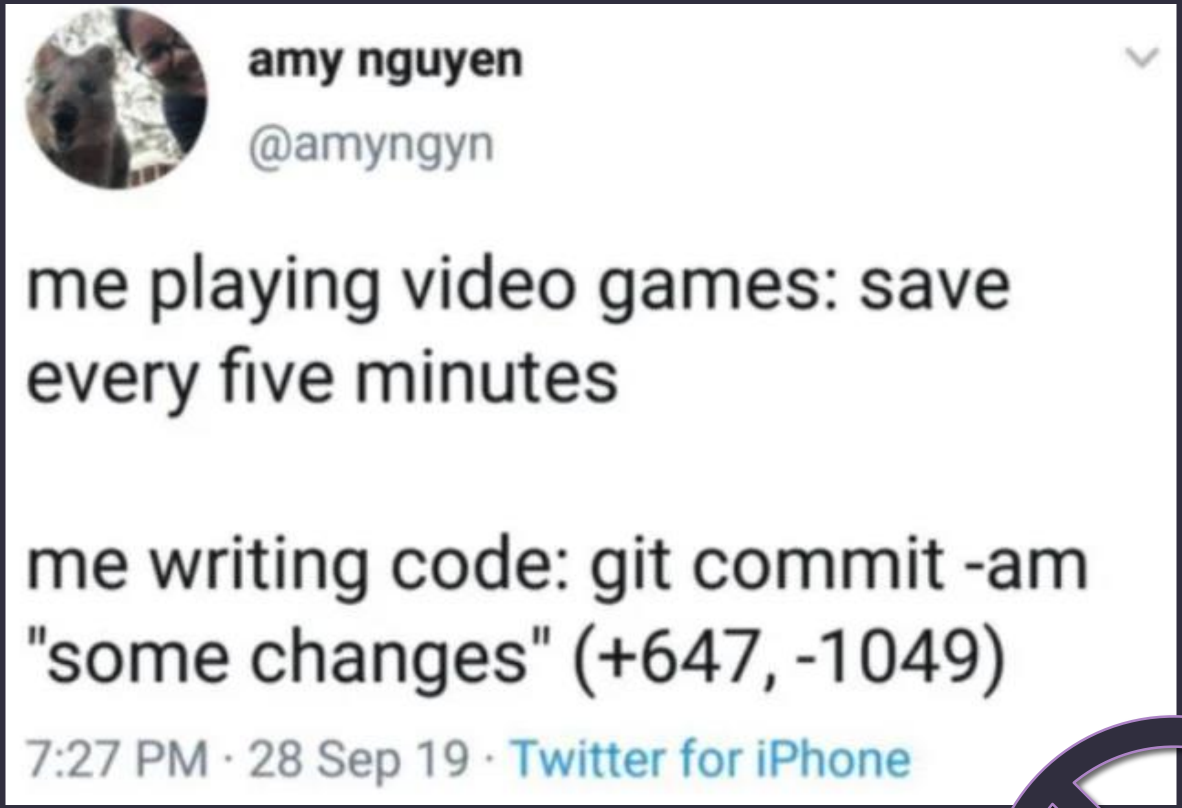
- The command `git status` shows the status of your current git state

- You then "add" your files to the staged state with the command `git add <filename>` or `git add *.txt` or `git add .` (everything)

- To move your staged files into a committed state, simply `git commit` (an editor will open for a commit message)

git basics: commit descriptions & frequency

- After `git commit` you get an editor to write a description of the commit
  - The first line of this description is called the "commit message"
  - Alternatively, you can do `git commit –m "commit message"`
- The message is the most important part of the description: get it right!
  - Must be short, to the point, and not contain useless information like commit author, date, branch, etc. (all of these can be found with the formal tools of git)
- If you want a description besides the main message, leave an empty line after the first, and then write as much text as you wish
- In other git environments (git GUIs, GitHub, …) there is a separate textbox for the message and the rest of the description already in the GUI
- ***Commit regularly!***
  - Any change that you can summarize in a sentence should be a commit!
  - Need several sentences to summarize a change in code?
    Probably you should have committed more often!



amy nguyen
@amyngyn

me playing video games: save every five minutes

me writing code: git commit -am "some changes" (+647, -1049)

7:27 PM · 28 Sep 19 · Twitter for iPhone

When you commit after a long time and don't remember what you've done

```
git commit -m "some changes"
```

NO!

git basics: amend, reset, blame

- Sometimes you commit something but immediately notice a “typo” or a “one-liner” change. You can immediately add a change to the existing commit using `git amend`

- The main strength of keeping all this history is resetting back to a previous stage!
- With `git reset <commit>` you bring your code base back into the selected commit!
  - With flag `--soft`: makes reverted differences unstaged changes
  - With flag `--hard`: deletes all differences versus reverted stage
  - See documentation on git revert online for many more possibilities!

- Git allows you to find out who, and when, edited a file with `git blame <file>`
  - used by bad people that like to blame other people
  - JUST KIDDING, extremely useful tool: author of change = most likely to know why change

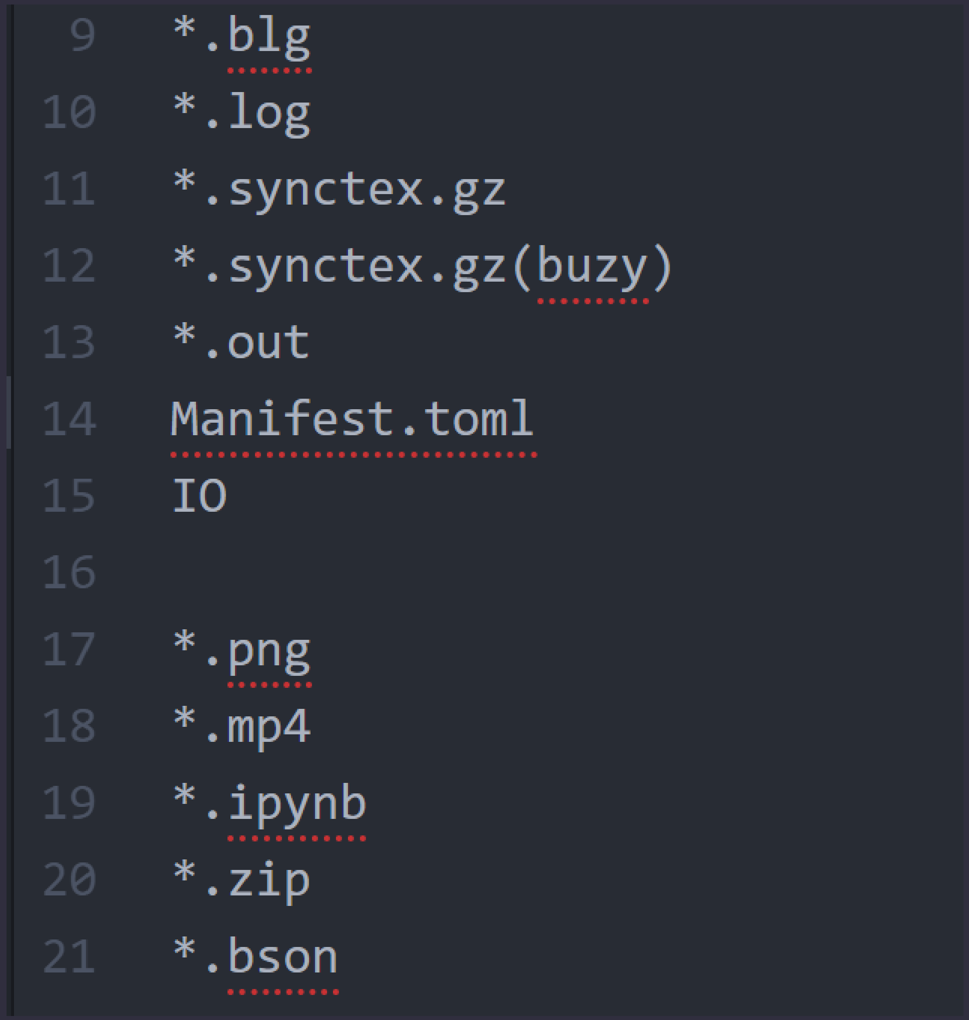
git reset --hard

Exercise: git basics

1. If git isn't installed on your machine, do it
2. Create a local folder and in it initialize a git repository: `git init`
3. Configure your user name and email
   - `git config --global user.name “My Name”`
   - `git config --global user.email name@example.com`
4. Create a file “test.txt” and in it write something in the 1st line only.
5. Use `git status` and see that indeed the file pops up as modified
6. Stage and commit the file with appropriate commit message!
7. Now edit the “test.txt” and add a 2nd line of text. Commit the change.
8. Oh noes, your change was really bad! Reset to previous commit!
   1. First, do a soft revert. Change the final character of the second line into ‘a’. Commit. Now you have a version of the “test.txt” that has 2 lines, the second ending in ‘a’.
   2. Now, revert again, but do a hard revert. You lost everything in the 2nd line. That's okay.

git basics: file size warning!

- ❑ git is an excellent tool for text-based files
- ❑ git is a bad tool for binary files, videos, etc.
- ❑ Since nothing is ever deleted from your history if you add a large file to your git commit, this file will forever burden your repository
- ❑ Plan what you really need to version, and add things you don’t want to version into the .gitignore file! Things there are untracked



```
*.blg
*.log
*.synctex.gz
*.synctex.gz(buzy)
*.out
Manifest.toml
IO

*.png
*.mp4
*.ipynb
*.zip
*.bson
```

git basics: tags, branches, and merging

- Tags allow you to “mark” a specific commit with a special name. You can revert to tags just as well as to any other commit. E.g. tag “submitted” for the version of the code submitted for publication

- You can also make several ***branches*** of your code base with `git branch <name>`
  - Branches are different instances of the code base that can be developed independently
  - You can jump from one branch to another at any point using `git checkout branch-name`
  - One branch is always `main`, to which all other development branches are merged into

- A branch can be ***merged*** into another branch, which simply “adds” the changes tracked by one branch to the other
  - If both branches contain commits that state changes on the same file+line: ***merge conflict***
  - The conflict must be manually resolved by a human which decides which change survives

GIT PUSH
MY PROJECT
GIT PUSH
GIT MERGE CONFLICT
imgflip.com

Different ways to merge

1. Merge commit: all history kept + 1 extra commit for “merging”
2. Squash merge: no history retained, all changes become a single commit
3. Rebase: all commits individually added 1 by 1 to the merging branch

- ❑ Live showcase at https://git-school.github.io/visualizing-git/

- ❑ DataFrames.jl maintainers recommend: ***always squash***
  - In most pull requests the history of the final “product” doesn’t matter. Only the product.
  - Makes it much easier to cherry pick changes (cherry-picking not discussed here)
  - Rebasing in collaborative projects will bring in countless unnecessary commits very quickly

- ❑ Rebasing can be useful in solo projects or with small teams!
  - Retaining full history can be used to justify choices made or better blame people!

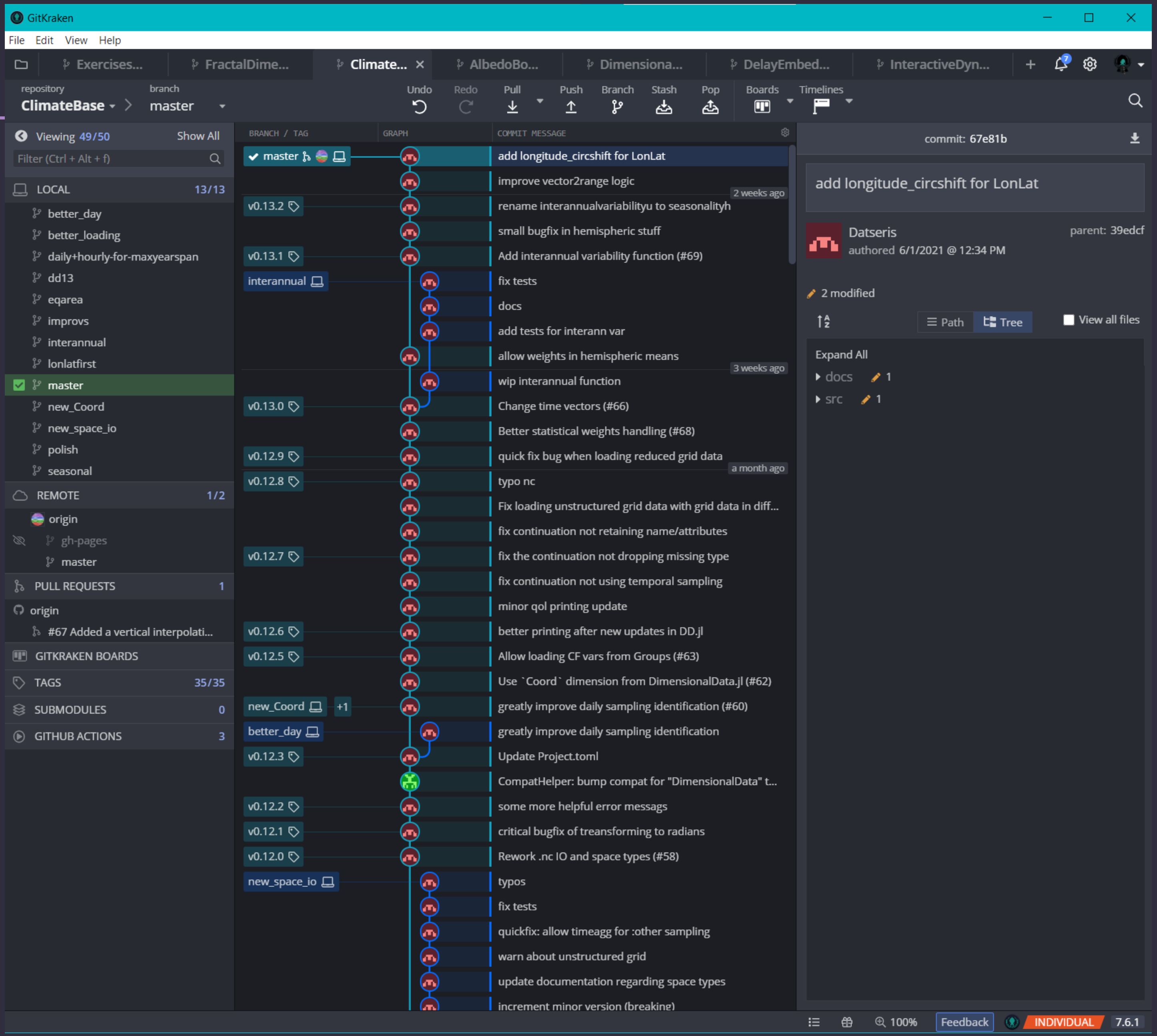
- ❑ In every case best practice is to keep feature branches ***small and focused!***

Exercise: git merge & conflict

- Now you are in branch `main`. Create a second branch `bad` and switch to it
- Add a 2nd line to `test.txt`. Commit.
- Checkout back to `main` branch. 2nd line doesn't exist. Create third branch `feature`.
- In `feature`, open again `test.txt` and add a ***different*** 2nd line. Commit.
- Now merge `feature` into `main`. Now 2nd line of `test.txt` is also in `main`!
- --- Optional part: conflict management (total exercise time: 30 minutes)
- Then, switch to `bad`. The 2nd line that was in `main` is now different, and is the "old" 2nd line you have added in `bad`.
- Try to merge `bad` into `main`, but you'll see it is not possible: Conflict!
- Conflicts must be resolved explicitly in an editor (that e.g. git provides)
- Resolve the change by making the 2nd line of `bad` the 3rd line of the file. Then merge `bad` into `main`.
- By the end you should have a `test.txt` with 3 lines of text!

GitKraken

- ❑ GitKraken is an exceptional GUI based git manager
  - Free for local projects
  - Free for remote projects
  - Not free for remote & private!
- ❑ Absolutely spectacular!!!
  - I'm not endorsed by GitKraken
- ❑ VSCode has also very good integrated git support (fully free) but its timeline view isn't as nice as GitKraken



BRANCH / TAG	COMMIT MESSAGE	
master	add longitude_circshift for LonLat	
	improve vector2range logic	2 weeks ago
v0.13.2	rename interannualvariabilityu to seasonalityh	
	small bugfix in hemispheric stuff	
v0.13.1	Add interannual variability function (#69)	
interannual	fix tests	
	docs	
	add tests for interann var	
	allow weights in hemispheric means	3 weeks ago
	wip interannual function	
v0.13.0	Change time vectors (#66)	
	Better statistical weights handling (#68)	
v0.12.9	quick fix bug when loading reduced grid data	a month ago
v0.12.8	typo nc	
	Fix loading unstructured grid data with grid data in diff...	
	fix continuation not retaining name/attributes	
v0.12.7	fix the continuation not dropping missing type	
	fix continuation not using temporal sampling	
	minor qol printing update	
v0.12.6	better printing after new updates in DD.jl	
v0.12.5	Allow loading CF vars from Groups (#63)	
	Use `Coord` dimension from DimensionalData.jl (#62)	
new_Coord +1	greatly improve daily sampling identification (#60)	
better_day	greatly improve daily sampling identification	
v0.12.3	Update Project.toml	
	CompatHelper: bump compat for "DimensionalData" t...	
v0.12.2	some more helpful error messages	
v0.12.1	critical bugfix of transforming to radians	
v0.12.0	Rework .nc IO and space types (#58)	
new_space_io	typos	
	fix tests	
	quickfix: allow timeagg for :other sampling	
	warn about unstructured grid	
	update documentation regarding space types	
	increment minor version (breaking)	

commit: 67e81b

add longitude_circshift for LonLat

Datseris
authored 6/1/2021 @ 12:34 PM
parent: 39edcf

2 modified

Your code → re-traceable & safe

- ❑ Initialize a git repository for your code.
  - Make sure default branch is `main` instead of `master` if you are using an old version of git
  - Use `git branch -M <oldname> <newname>` if necessary
- ❑ Think of the .gitignore file, and appropriately ignore files that are part of your code but should not be tracked (e.g., binary files, plots, input data, …)
- ❑ Add to the git repository the initial version of the code
- ❑ Commit all files, and then create a tag on this new commit
  - Name suggestions: shame, neverlookback, pliz_help_me, a_star_is_born, …

- ❑ From now on, all exercises on your own code base must be solved by making a branch for the exercise solution, developing the solution on the branch, and then merging the branch into `main`!