Making your life easier with git

Jakob van Santen
Zeuthen Data Science Seminar, 2022-02-17

HELMHOLTZ

DESY.

Who is this talk for?

You, if:

- You have never heard of git.
- You have heard of git, and know you should be using it, but still keep files named analysis.py, analysis-copy.py analysis-copy1.py, analysis-copy1-reallyfixedthistime.py, etc.
- You use git, but are overcome with dread every time you hit a merge conflict.
- You are a git ninja, and want to argue with me over `git pull --rebase`.

Things you should get out of this talk:

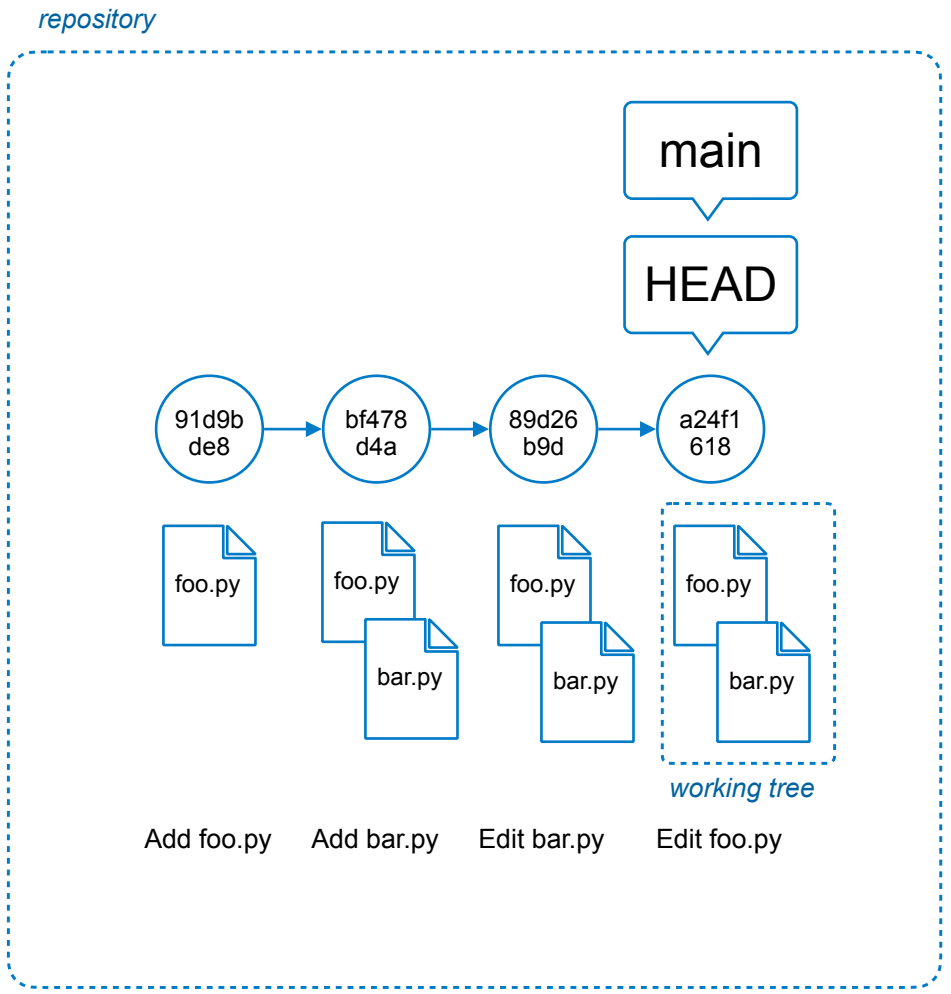
- Why version control is important
- How to use basic git commands
- How to write useful commit messages
- What `git merge` actually does, and how to face merge conflicts without fear
- A collection of useful commands to try out

What is git?

- Git is a version control system, a tool to track changes (mostly to text files).
- Git has been the standard version control system for ~5-10 years
- Tracking changes (with git) is good for:
  - **you, today**:
    - Know exactly what you changed since lunchtime.
    - Never make negative progress. If you break everything, at the end of the day you can always revert to the version you had this morning.
  - **you, in the future**: why does my code behave differently than it did a month ago? What was I thinking when I made this change?
  - **your colleagues**: how *exactly* is my script different from yours?

git jargon

- **repository** (*noun*): a group of files (and their histories) that are tracked together. Can be a single file, or 3.5M (Windows).
- **commit** (*noun*): a snapshot of the files in the repository, along with a note and links to one or more parent commits. Identified by a hash of its contents.
- **working tree** (*noun*): set of files on the filesystem, some of which may not be tracked by git.
- **checkout** (*verb*): extract the contents of a given commit to the working tree
- **ref** (*noun*): a short-hand name for a specific commit (hash)
- **HEAD** (*noun*): ref for the currently checked-out commit
- **branch** (*noun*): a ref (name) for the latest in a chain of commits. The default branch is usually called "main." Adding a new commit to a branch updates the ref.



Single-user git

Workflow: committing changes

After editing files tracked in a git repository,

- Step 0: determine what has changed, and decide what you want to commit
- Step 1: stage files for commit
- Step 2: compose a [useful] commit message and commit

- **diff** (*noun*): difference between two snapshots, expressed as lines added (+) and lines removed (-)
- **stage** (*verb*): `git add` marks a file from the working copy for inclusion in the next commit
- **commit** (*verb*): `git commit` adds a new commit to the current branch.

Step 0: decide what to commit

git status

`git status` tells you

- Which branch you currently have checked out
- How that branch relates to any remote-tracking branches
- Which files in the working tree have uncommitted changes
- Which files in the working tree are not tracked at all

```
(base) jakob@znb140 gen2-analysis % git status
On branch master
Your branch is up to date with 'origin/master'.

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:   .gitignore
    modified:   conda-lock.yml
    modified:   figures/toise.mplstyle
    modified:   toise/figures/pointsource/flare.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    5eff16a895f6287eeaf9674e60d751a9/
    notebooks/data preparation/Neutrino physics.ipynb
    pyproject.toml.bak
    toise/notebooks/

no changes added to commit (use "git add" and/or "git commit -a")
```

Step 0: decide what to commit

git diff

`git diff` shows you the difference between two snapshots (with no arguments, these are HEAD and the working copy)

- +: line was added
- -: line was removed
- Changed lines are denoted by a deletion (-) followed by an addition (+)
- `git diff --word-diff` shows differences within a line
- `git diff --stat` shows you how many lines were changed per file

```
diff --git a/.gitignore b/.gitignore
index 620e424..70009bb 100644
--- a/.gitignore
+++ b/.gitignore
@@ -4,3 +4,7 @@
 data/
 figures/**/*.json.gz
 figures/**/*.pdf
+notebooks/**/*.pdf
+notebooks/**/*.fits
+notebooks/**/.ipynb_checkpoints
+.vscode
diff --git a/conda-lock.yml b/conda-lock.yml
index 571c538..6e31e54 100644
--- a/conda-lock.yml
+++ b/conda-lock.yml
@@ -37,828 +37,690 @@ package:
   url: https://conda.anaconda.org/conda-forge/linux-64/_libgcc_mutex-0.1-
conda_forge.tar.bz2
   version: '0.1'
 - category: main
-  dependencies:
-    _libgcc_mutex: 0.1 conda_forge
-    libgomp: '>=7.5.0'
```

```
(base) jakob@znb140 gen2-analysis % git diff --stat
 .gitignore                          |    4 +
 conda-lock.yml                      | 7580 ++++++++++++++++++++----------------
 figures/toise.mplstyle              |    2 +
 toise/figures/pointsource/flare.py |    1 +
 4 files changed, 4166 insertions(+), 3421 deletions(-)
```

Step 1: prepare a commit

git add

`git add` marks a file from the working tree for inclusion in the next commit

- `git diff` no longer shows changes
- `git diff --staged` shows staged changes
- Edits you make after this point are not staged unless you explicitly `git add` again.
- Stage parts of files with e.g. `git add --patch <file>`

```
(base) jakob@znb140 gen2-analysis % git add .gitignore
(base) jakob@znb140 gen2-analysis % git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   .gitignore

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:   conda-lock.yml
    modified:   figures/toise.mplstyle
    modified:   toise/figures/pointsource/flare.py
```

```
(base) jakob@znb140 gen2-analysis % git diff --stat
 conda-lock.yml                      | 7580 ++++++++++++++++++++----------------------
 figures/toise.mplstyle              |    2 +
 toise/figures/pointsource/flare.py |    1 +
 3 files changed, 4162 insertions(+), 3421 deletions(-)
(base) jakob@znb140 gen2-analysis % git diff --stat --staged
 .gitignore | 4 ++++
 1 file changed, 4 insertions(+)
```

Step 2: compose a commit message

git commit

`git commit` commits the staged snapshot to the repository

- By default, `git commit` dumps you into your default editor (usually vi) to compose a commit message. Set EDITOR in your environment if you prefer a different editor.
- Alternatively, provide a commit message directly with -m, e.g. `git commit -m "Ignore notebook products"`
- `git diff --staged` shows staged changes

```
(base) jakob@znb140 gen2-analysis % git commit .gitignore
[master ac305d3] Ignore notebook products
 1 file changed, 4 insertions(+)
```

```
Ignore notebook products
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Your branch is up to date with 'origin/master'.
#
# Changes to be committed:
#       modified:   .gitignore
#
# Changes not staged for commit:
#       modified:   conda-lock.yml
#       modified:   figures/toise.mplstyle
#       modified:   toise/figures/pointsource/flare.py
#
# Untracked files:
#       5eff16a895f6287eeaf9674e60d751a9/
#       notebooks/data preparation/Neutrino physics.ipynb
#       pyproject.toml.bak
#       toise/notebooks/
#
~
~
~
-- INSERT --
```

Viewing history

git log

`git log` shows the commit history

- By default, `git log` lists history from HEAD backwards. Pass a specific ref to start there instead.
- `git log --oneline` shows only the subject line of each commit
- `git log --oneline --graph` shows branches
- `git show <ref>` shows the diff between <ref>'s parent commit and <ref>, e.g. `git show HEAD` shows the diff of the last commit.

For the git ninja: what does `git show` do with a merge commit?

```
(base) jakob@znb140 gen2-analysis % git log
```

```
commit 18a82ec2e3fc9697b474dc6bb1728bbd6d15239d (HEAD -> master)
Author: Jakob van Santen <jvansanten@gmail.com>
Date:   Thu Feb 10 11:20:16 2022 +0100

    Ignore notebook products

commit 08587c2b49933c305ee0d030af61734aa2553f51 (origin/master, origin/HEAD)
Author: Jakob van Santen <jvansanten@gmail.com>
Date:   Thu Jan 27 15:45:59 2022 +0100

    Add license, with pointers to bundled components

commit c691b1f825c478888604b5aa67d1a0af62bc33bf
Author: Jakob van Santen <jvansanten@gmail.com>
Date:   Thu Jan 27 14:13:05 2022 +0100

    Make duration settable in flare plot

commit efe4fddc73b4a62d8a41baa7a1b840068a773d60
Author: Jakob van Santen <jvansanten@gmail.com>
Date:   Fri Jan 14 16:11:00 2022 +0100

    Use annotations for fancier cli

:
```

Useful commit messages

https://xkcd.com/1296 CC BY-NC 2.5

Commit messages should tell the reader (in order):

- *What* a change does
- *Why* it is necessary
- *How* it works

There are a lot of rules floating around. How strictly you follow them depends on whether you're writing for you-in-the-future or collaborators on a large, structured project.

Exercise: read commit messages you wrote 2 months ago. Can you reason about what each commit does from the commit message alone?

```
commit bf478d4a4fd82dfe6d23c4b4ada0aeef3023819c
Author: Redacted Ralph <YYYYYYYYYYY@gmail.com>
Date:   Fri Jan 21 15:06:11 2022 +0100

    UnitLoader: resolve secrets for AmpelBaseModel subclasses

    AmpelBaseModel subclasses have direct annotations in __annotations__,
    but annotations of superclasses in _annots.  Look for secrets in
    _annots to allow secret fields to be defined in base classes, even
    with multiple inheritance.
```

Incremental commits

Sometimes you can't finish a change in one sitting, or compose the perfect commit message.

- Commit often, while you still remember what a change means. Avoid leaving changes in your working copy at the end of the day.
- Git lets you combine and rewrite commits after the fact
  - `git commit --amend` updates (replaces!) the last commit
  - `git rebase -i HEAD~5` edits the last 5 commits
- Never change commits that have been pushed to a remote!

```
(base) jakob@znb140 gen2-analysis % git rebase -i HEAD~5
Successfully rebased and updated refs/heads/master.
```

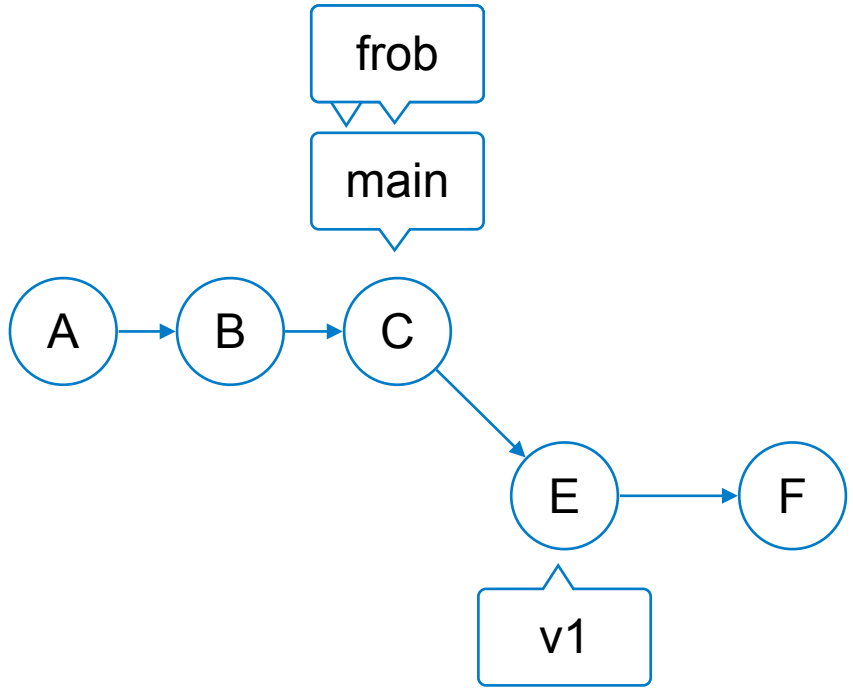
```
pick e0c2987 make uniform how units are put in figures, [ vs (, and also, make
sure the number of radio stations is 30 everywhere
pick efe4fdd Use annotations for fancier cli
pick c691b1f Make duration settable in flare plot
pick 08587c2 Add license, with pointers to bundled components
pick 18a82ec Ignore notebook products

# Rebase e135507..18a82ec onto e135507 (5 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
<Cube/projects/2021/gen2-analysis/.git/rebase-merge/git-rebase-todo" 30L, 1417B
```

Branching, tagging, merging

Branches are cheap. Use a branch for any change you don't want to land on main *right now.*

- **branch** (*noun*): a ref (name) for the latest in a chain of commits. You can have an arbitrary number of branches. The default branch is usually called "main." Adding a new commit to a branch updates the ref.
- **tag** (*noun*): a permanent ref (name) for a specific commit. Usually used to mark a version that you want to keep for a while (a “release”). New commits do not update a tag.




```
git checkout -b frob
git commit -m "add freebs" freebs.txt
git tag v1
git commit -m "add froobles" froobles.txt
git checkout main
git merge frob
```

NB: since there is a direct path from C to F, git merges by moving the main ref (a fast-forward merge).

3-way merges and conflict resolution

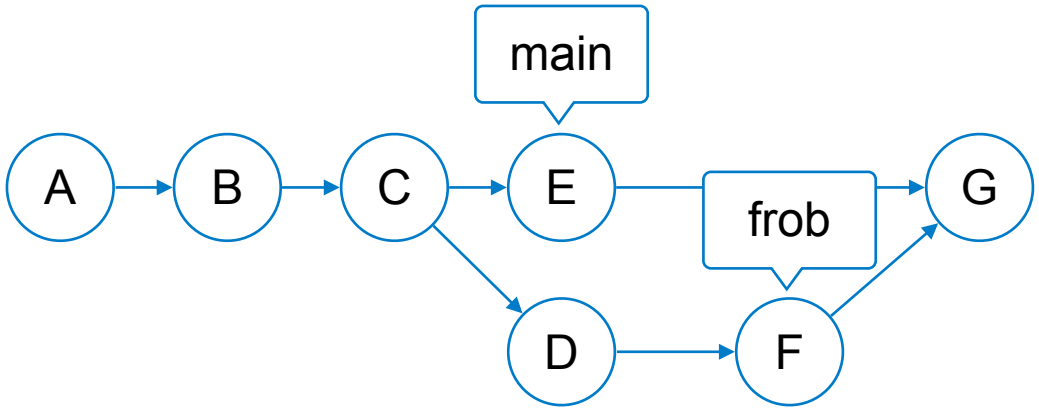
If anyone committed to main since frob diverged, git performs a 3-way merge, using E, F, and their nearest common ancestor C.

git calculates 2 diffs:

- *ours*: diff C E
- t*heirs*: diff C F

and applies them in parallel:

- If both *ours* and *theirs* contain the same changes, apply one
- If *ours* touches a line that *theirs* leaves unchanged, apply *ours* (and vice versa)
- If ours and theirs apply different changes, you have a merge conflict. Don't panic!



Conflict resolution

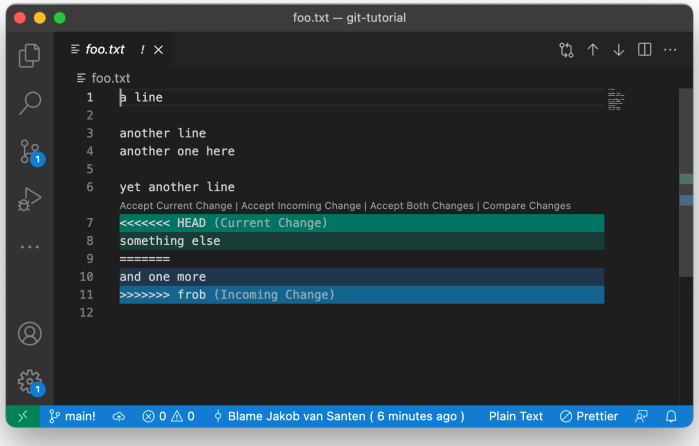
Merge conflicts have to be resolved manually. Keep *ours* and *theirs* straight and you'll be fine.

- Accept one side of the conflict with `git checkout --ours <file>` or `git checkout --theirs <file>`
- Alternatively, edit the file to keep only the side of each conflict you want to keep
- Some IDEs/clients have tools to make this easier

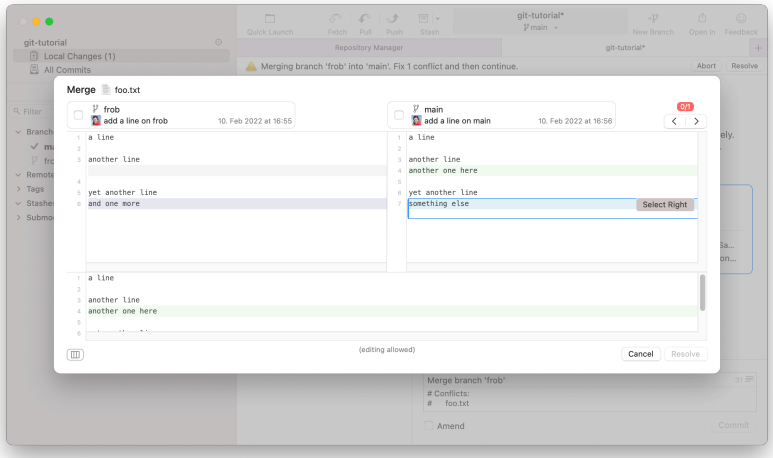
Advanced Merging in the Git Book has more strategies for untangling merge conflicts.

The longer a branch exists, the more merge conflicts it can accumulate. Keep the most common ancestor recent by rebasing onto or merging the base branch.

```
(base) jakob@znb140 git-tutorial % git merge frob
Auto-merging foo.txt
CONFLICT (content): Merge conflict in foo.txt
Automatic merge failed; fix conflicts and then commit the result.
```



Merge conflict highlighting in VSCode



Merge tool in Fork

NB: a clean merge does not mean that the resulting code is valid, text makes sense, etc., only that the diffs from each side did not collide. Test a merge commit just like any other, and amend as necessary.

Stashing changes

`git stash` shunts all changes (both staged and unstaged) into a temporary commit and resets the working tree to HEAD. This can be useful (sometimes, necessary) before commands that change HEAD (e.g. checkout, rebase).

- `git stash pop` applies last stash to the working tree
- `git stash list` lists the stack of stashes
- `git stash push <file1> <file2>` stashes only the specified files
- `git stash push --patch` stashes parts of files

```
(base) jakob@znb140 gen2-analysis % git status -s
 M conda-lock.yml
 M figures/toise.mplstyle
 M toise/figures/pointsource/flare.py
(base) jakob@znb140 gen2-analysis % git stash
Saved working directory and index state WIP on master: 18a82ec Ignore notebook
products
(base) jakob@znb140 gen2-analysis % git status -s
(base) jakob@znb140 gen2-analysis % git stash pop -q
(base) jakob@znb140 gen2-analysis % git status -s
 M conda-lock.yml
 M figures/toise.mplstyle
 M toise/figures/pointsource/flare.py
```

When things go wrong

~~Nuke it from orbit~~ git reset

The reference log (reflog) records when references (HEAD, main, etc) changed. Use it to un-do commits [that you haven't pushed yet].

- `git reset --soft HEAD@{1}` sets the current branch tip to where it was 1 operation ago, but leaves the last state in the working tree.
- `git reset --hard HEAD@{5}` sets the current branch tip, staging area, and work tree to where they were 5 operations ago.
- `git reset --mixed main@{yesterday}` sets the current branch tip to where main was yesterday, and stages any changes w.r.t. that commit.

```
(base) jakob@znb140 Ampel-core % git reflog
```

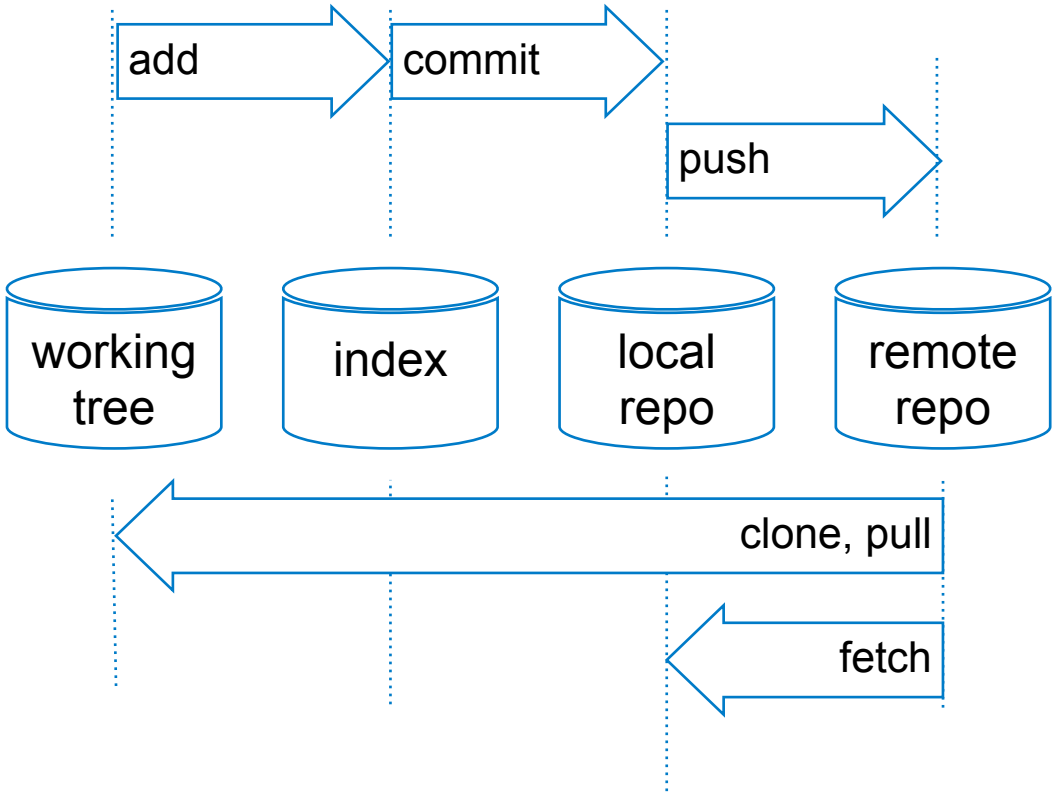
```
156bbef3 (HEAD -> dev/v0.8.2, tag: v0.8.2-alpha.10, origin/dev/v0.8.2)
HEAD@{0}: commit: Bump version
bd1ee73f HEAD@{1}: rebase (finish): returning to refs/heads/dev/v0.8.2
bd1ee73f HEAD@{2}: rebase (pick): test: server reload_config
1ae3ecf2 HEAD@{3}: rebase (pick): AbsOpsUnit: mark logger as traceless
518e5822 HEAD@{4}: rebase (pick): UnitLoader: do not mutate UnitModel on
validation
f9b8fd3b HEAD@{5}: rebase (fixup): AbsEventUnit: mark process_name traceless
6027a307 HEAD@{6}: rebase (start): checkout
91d9bde8b8cc30f4b8d9387217bbaabe249990e9
9cc8e1a4 HEAD@{7}: commit: test: server reload_config
6a21985e HEAD@{8}: commit: fixup! AbsEventUnit: mark process_name traceless
cba45cf7 HEAD@{9}: commit: AbsOpsUnit: mark logger as traceless
e927e881 HEAD@{10}: commit: UnitLoader: do not mutate UnitModel on validation
6027a307 HEAD@{11}: commit: AbsEventUnit: mark process_name traceless
91d9bde8 (tag: v0.8.2-alpha.9) HEAD@{12}: rebase (finish): returning to refs/
heads/dev/v0.8.2
91d9bde8 (tag: v0.8.2-alpha.9) HEAD@{13}: rebase (start): checkout HEAD~5
91d9bde8 (tag: v0.8.2-alpha.9) HEAD@{14}: reset: moving to HEAD
91d9bde8 (tag: v0.8.2-alpha.9) HEAD@{15}: commit: Bump version
bf478d4a HEAD@{16}: commit (amend): UnitLoader: resolve secrets for
AmpelBaseModel subclasses
01513bbf HEAD@{17}: commit (amend): UnitLoader: resolve secrets for
AmpelBaseModel subclasses
:
```

Multi-user git

More jargon

- **init** (*verb*): create an empty repository
- **remote** (*noun*): name for another repository (directory or URL) that you can exchange commits with. The default remote is "origin." For forked repositories, it is common to add the repository you forked from as "upstream."
- **clone** (*verb*): create a local copy of a remote. The cloned repository is independent of the remote, and in principle co-equal.
- **fetch** (*verb*): download new commits from a remote
- **pull** (*verb*): fetch and integrate commits from remote
- **push** (*verb*): add commits local commits to remote
- **fork** (*noun*): a (hosted) clone of a repository. Fork + PR is a common way to allow collaboration without direct commit access.

Moving content between git's different areas



Repository hosting

Who owns your remote?

Free for personal use

~Everyone has a GitHub account

Free-tier repositories ~have to be public

Your personal repositories are associated with you, personally. Not great for discoverability or long-term preservation.

What happens when Microsoft decide they need GitHub to actually make money?

(gitlab.desy.de)

Hosted at DESY

Everyone at DESY has a GitLab account, external users can get one via Helmholtz AAI (eduGAIN/ORCID/GitHub/Google)

~Unlimited private repositories, CI/CD pipelines, package hosting, etc.

Can use groups to organize related repositories for better discoverability and preservation

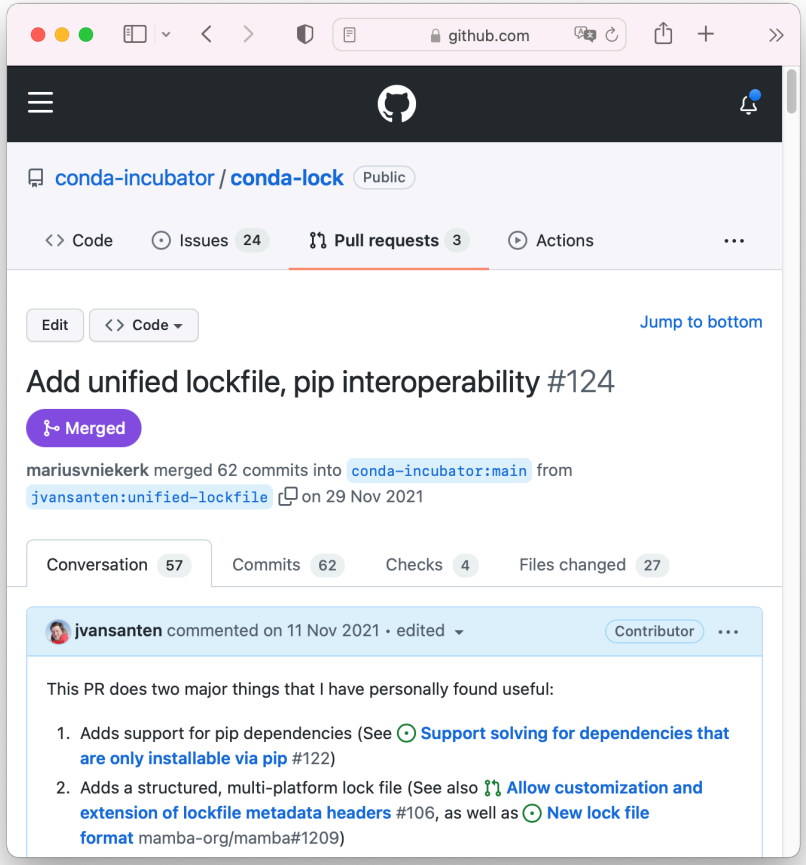
Otherwise, similar features. Consider using DESY GitLab for new work.

Collaboration workflows

Pull/Merge Requests let you propose, test, and discuss changes before integrating them. How heavily you use them depends on how large and complicated your project is.

- **Commit directly to main** if you have <= 3 collaborators, no automated testing, and no non-developer users
- **Make a branch in the same repo and create a PR** if you have > 3 collaborators, or automated testing in place, or have non-developer users
- **Make a branch in a fork of the repo and create a PR** if your collaborators are not known *a priori*

A Pull Request on GitHub



Rejected pushes

`git push` will fail if the remote branch contains commits that the local branch does not.

- This is normal when working on a branch that other people commit to.
- `git pull --rebase` and try again.

```
(base) jakob@znb140 gen2-analysis % git push
To github.com:icecube/gen2-analysis.git
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'github.com:icecube/gen2-analysis.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Rebase resolves conflicts the same way as merge, but with the meaning of "ours" and "theirs" reversed.

Sometimes you need to force-push, e.g. after rebasing a branch that only you work on. Never, ever force-push to a branch that someone else could have seen in the mean time. It is a good idea to to turn on branch protections (GitHub, GitLab) to prevent yourself or anyone else from force-pushing accidentally.

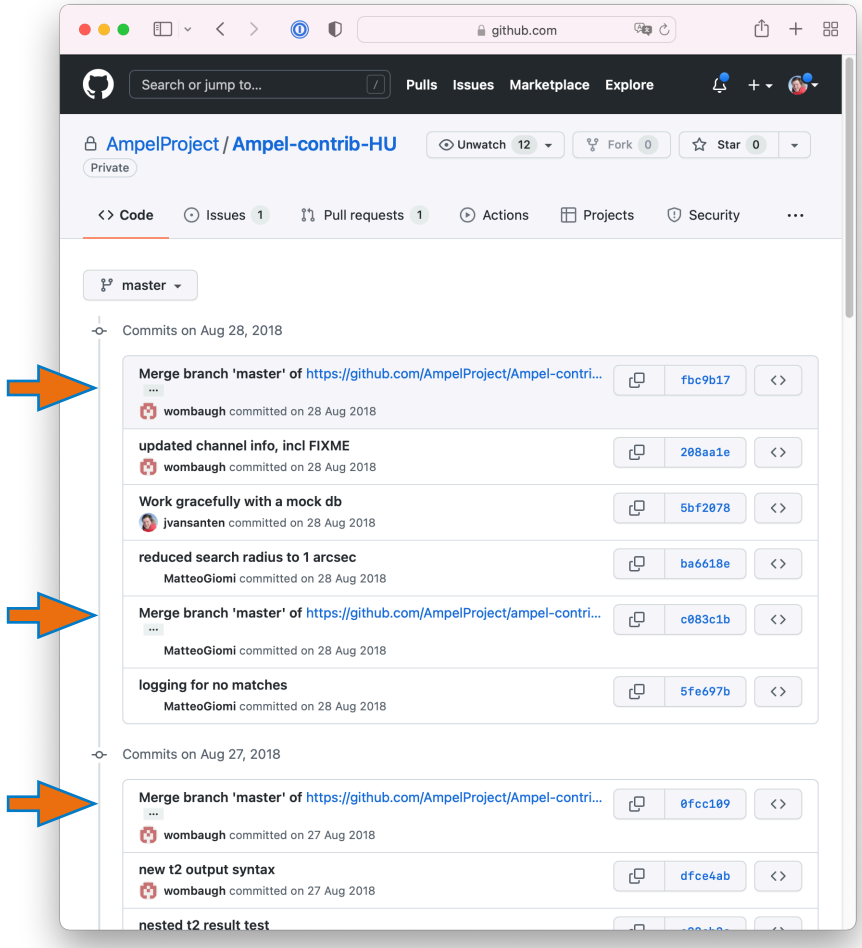
Keep your history readable

with `git pull --rebase`

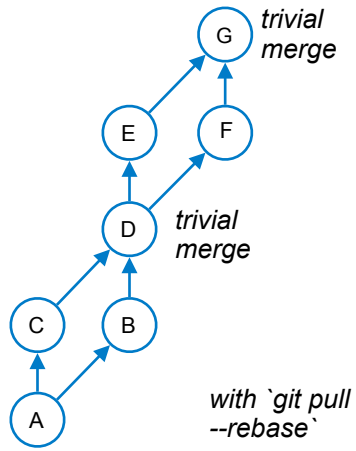
`git pull` merges origin/main with main by default. This is rarely what you actually want.

- `git pull --rebase` reapplies local commits on top of the remote branch tip, creating a linear history
- Reserve merges for branches that diverged on purpose.
- Set the correct default with `git config --global pull.rebase true`
- Corollary: on a forked repository, keep main identical to upstream/main. Never commit directly to the default branch.

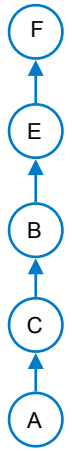
History cluttered with unnecessary merge commits



with `git pull`



with `git pull --rebase`



It is surprisingly hard to prevent trivial merge commits if you let people push directly. The closest you can get is to protect your default branch and require PRs to be closed via squash or rebase.

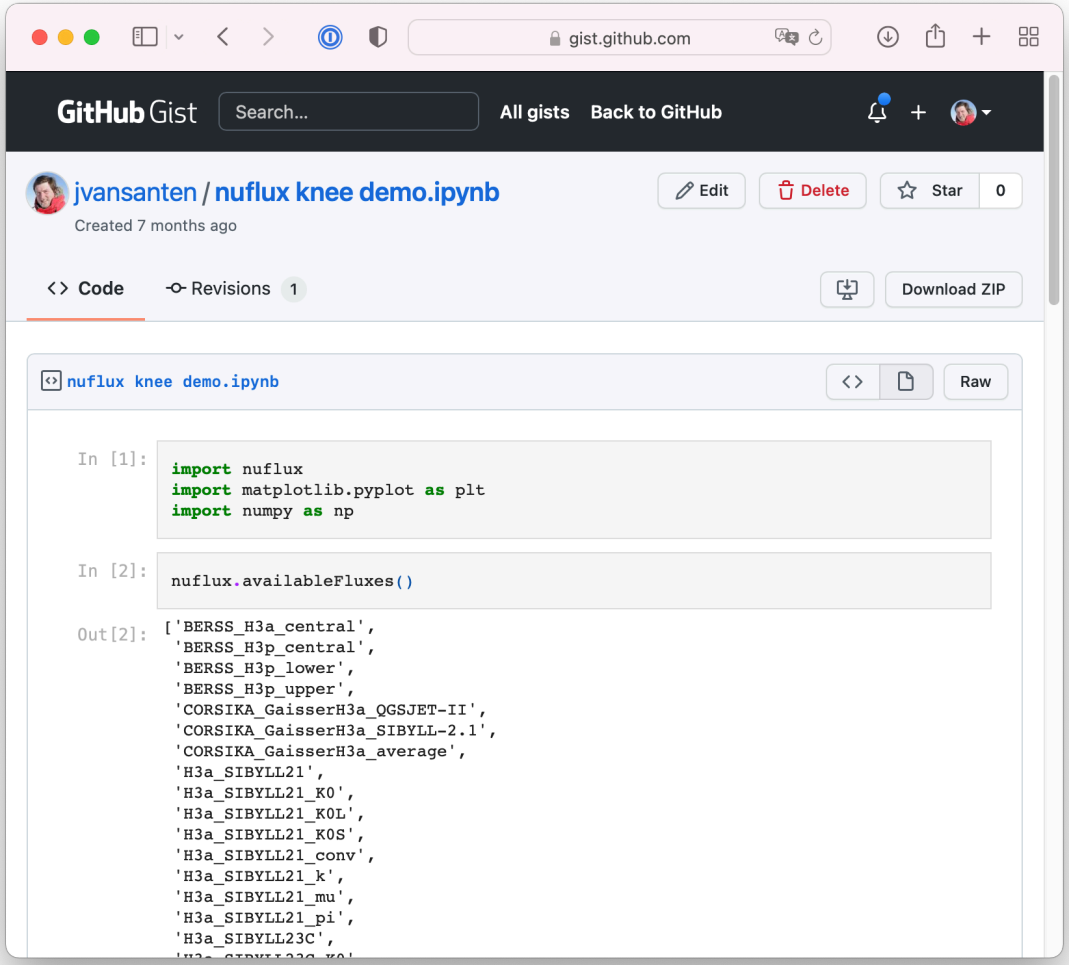
Tidbits

Single-use repositories

Whenever you get a random tar archive that you want to change, commit its contents to a throw-away repository so you can track exactly what you change.

```
(base) jakob@znb140 sncosmo % git init
Initialized empty Git repository in /Users/jakob/Downloads/sncosmo/.git/
(base) jakob@znb140 sncosmo % git add .
(base) jakob@znb140 sncosmo % git commit -m 'initial commit'
[main (root-commit) a7c3642] initial commit
 19 files changed, 51993 insertions(+)
 create mode 100644 ZTF18aajjhkq_salt2_NEDz_fit.svg
 create mode 100644 ZTF18abwitkf_salt2_NEDz_fit.svg
 create mode 100644 ZTF21aaaadmo_salt2_NEDz_fit.svg
(base) jakob@znb140 sncosmo % vi ZTF18aajjhkq_salt2_NEDz_fit.svg
(base) jakob@znb140 sncosmo % git diff
diff --git a/ZTF18aajjhkq_salt2_NEDz_fit.svg b/ZTF18aajjhkq_salt2_NEDz_fit.svg
index 168711d..13f74e2 100644
--- a/ZTF18aajjhkq_salt2_NEDz_fit.svg
+++ b/ZTF18aajjhkq_salt2_NEDz_fit.svg
@@ -11,7 +11,7 @@
     <dc:format>image/svg+xml</dc:format>
     <dc:creator>
      <cc:Agent>
-      <dc:title>Matplotlib v3.3.3, https://matplotlib.org/</dc:title>
+      <dc:title>Matplotlib v3.3.8, https://matplotlib.org/</dc:title>
      </cc:Agent>
     </dc:creator>
    </cc:Work>
```

Use GitHub Gists / GitLab Snippets to share bits of code and text



Useful resources

Documentation, tutorials, and opinions

- Oh My Git: a game for learning git
- Git tutorials from Atlassian
- How to Write a Git Commit Message
- Git Pro: an exhaustive reference manual
- Plumbing and Porcelain: section on git internals
- Think like Git: great talk on git internals at PyData 2021
- What's a "detached HEAD" in Git?
- Regain Control of Branches with git rebase --onto
- Ours & theirs in merge vs. rebase
- Writing a proper GitHub issue
- Best Practices for Using GitHub Issues
- SSH key auth for GitHub and GitLab
- Why you should use a code formatter

GUI clients and IDEs

- Fork
- GitKraken
- Tower
- GitHub Desktop
- VSCode

Best practices

- Keep as much of your work as possible in git.
- Commit early, commit often. Fix up incremental commits as you go.
- Push *finished* work to a remote.
- Write useful commit messages that communicate the *what*, *why*, and *how* of a change (in that order).
- Keep branches short-lived. If you need long-lived branches, rebase or merge often to keep the common ancestor recent.
- `git config --global pull.rebase true` (do it now)
- Never commit to the default branch of a forked repository.

Thank you

Git glossary

- **repository** (*noun*): a group of files that are tracked together. Can be a single file, or 3.5M (Windows).
- **commit** (*noun*): a snapshot of the repository (files and contents), along with a note and links to one or more parent commits. Identified by a hash.
- **working copy** (*noun*): set of files on the filesystem, not necessarily tracked by git
- **checkout** (*verb*): extract the contents of a given commit to the working copy
- **ref** (*noun*): a short-hand name for a specific commit (hash)
- **HEAD** (*noun*): ref for the currently checked-out commit
- **branch** (*noun*): a ref (name) for the latest in a chain of commits. The default branch is usually called "main." Adding a new commit to a branch updates the ref.
- **tag** (*noun*): a permanent ref (name) for a specific commit. Usually used to mark a version that you want to keep for a while (a “release”). New commits do not update a tag.
- **diff** (*noun*): difference between two snapshots, expressed as lines added (+) and lines removed (-)
- **stage** (*verb*): `git add` marks a file from the working copy for inclusion in the next commit
- **commit** (*verb*): `git commit` adds a new commit to the current branch.
- **init** (*verb*): create an empty repository
- **remote** (*noun*): name for another repository (directory or URL) that you can exchange commits with. The default remote is "origin." For forked repositories, it is common to add the repository you forked from as "upstream."
- **clone** (*verb*): create a local copy of a remote
- **fetch** (*verb*): download new commits from a remote
- **pull** (*verb*): fetch and integrate commits from remote
- **push** (*verb*): add commits local commits to remote
- **fork** (*noun*): a (hosted) clone of a repository. Fork + PR is a common way to allow collaboration without direct commit access.
- **amend** (*verb*): edit the latest commit to change its message or content
- **rebase** (*verb*): transplant a range of commits from one base commit to another.
- **squash** (*verb*): combine a range of commits
- **fixup** (*noun*): a (usually small) commit that fixes problems introduced by some other commit. Should usually be squashed into that other commit.
- **cherry-pick** (*verb*): copy a single commit from one branch to another (precisely: apply the diff between a commit and its parent to the current branch)

Contact

Deutsches Elektronen-
Synchrotron DESY

www.desy.de

Name Surname
Department
E-mail
Phone