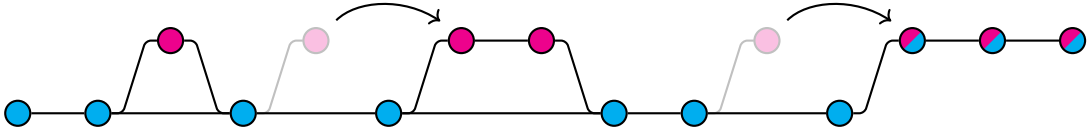
Git for Scientists

Outline

- Version Control
- What is Git?
- Git Basics
- Advanced Topics

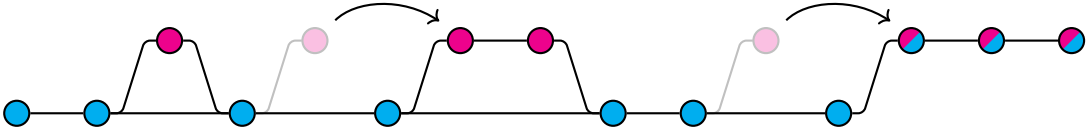


Version Control

- *"Well, it was working a minute ago."*
- Version control is a system for tracking changes in code/data.

- Keep track of changes made to project files.
- If mistakes are made, files can be reverted to any point in the history.

- **Worst case scenario:** keeping multiple copies of files or sending a zip file to every team member every time changes are made.
- Just... no.

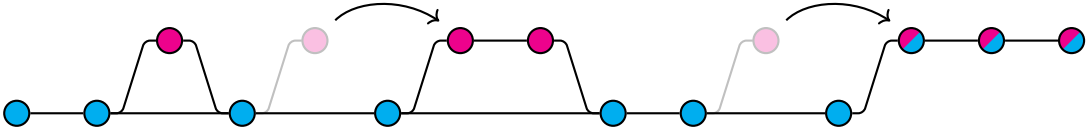


What is Git?

- Git is an application that helps ~~developers~~ scientists track files and their history using a distributed version control system.

- A historical record of changes to code & data.
- A structured way to collaborate on shared projects.

- *"Kind of like a shared digital lab notebook."*



Git Basics

- First, install Git.
  `https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`

```
user@host:~$ git --version
git version 2.30.2

user@host:~$ git config --global user.email "you@example.com"
user@host:~$ git config --global user.name "Your Name"
```

Repositories

- A repository (or "repo" for short) is the version control system for a project.
- Contains the history of changes made to files in your project.

Commits

- A "commit" is like a snapshot of the project.
- Every time changes are made to a project, we commit them to the repo.

The Initial Commit

- Say you have code in some directory `~/project`.
- We can initialize the repo using the `git init` command.

```
user@host:~$ cd project
user@host:~/project$ git init
Initialized empty Git repository in /user/project/.git/

user@host:~/project$ ls -A
.git  my_script.py  README.md
```

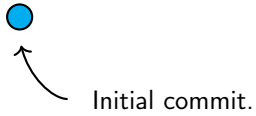
- It all starts with a commit.

```
user@host:~/project$ git commit -a -m "Initial commit."
2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README.md
 create mode 100644 my_script.py
```

- Here, the `-a` flag specifies that we want to include all files in our commit.
- We add a description to our commit using `-m "Initial commit."`

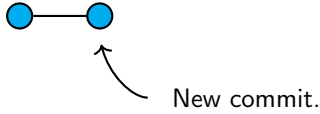
The Commit Graph

- We can represent the Git history at this point as a tree with a single node.



- As changes are made[1], we make new commits.

```
user@host:~/project$ edit my_script.py
user@host:~/project$ git commit my_script.py
```



- Note that the command `git commit <file>` only commits the changes to a single file. Other changed files will not be included as part of the commit.
- Each commit should have a meaningful description of the changes.

[1] We use `edit <file>` to denote changes made to a file.

Staging Changes

- We can selectively add files/changes to a commit.
- **Example:** adding & removing files from a commit, checking status.

```
user@host:~/project$ git add <file>
user@host:~/project$ git rm <file>
user@host:~/project$ git status
```

- Changes that are set to be included in a commit are called “staged” changes.

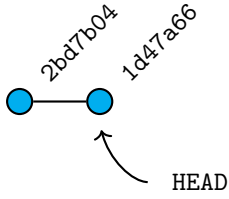
```
user@host:~/project$ edit my_script.py README.md
user@host:~/project$ git add my_script.py
user@host:~/project$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
modified:   my_script.py

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:   README.md
```

Viewing Commits

- The command `git status` is useful for checking what files are currently staged.
- The command `git log` lets us see the history of all commits to the project.

```
user@host:~/project$ git log --oneline
1d47a66 (HEAD -> main) Changed my_script.py
2bd7b04 Initial commit.
```



- Each commit has an associated ID (e.g. `2bd7b04` and `1d47a66`).
- The `HEAD` points to the current commit.

Undoing Changes

- If you commit too early and want to amend a previous commit.

```
user@host:~/project$ git commit -m "Initial commit."
user@host:~/project$ git add <forgotten_file>
user@host:~/project$ git commit --amend
```

- If you want to "un-modify" an unstaged file (change it back to the last commit).

```
user@host:~/project$ edit my_script.py
user@host:~/project$ git restore my_script.py
```

- If you want to "un-modify" a staged file (change it back to the last commit).

```
user@host:~/project$ edit my_script.py
user@host:~/project$ git add my_script.py
user@host:~/project$ git restore --staged my_script.py
```

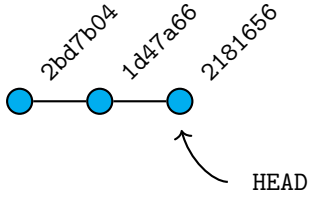
- If you want to temporarily view a previous version of the project.

```
user@host:~/project$ git checkout 2bd7b04
```

Undoing Commits

- Sometimes, we may want to go back to a previous state of the project.
- We can use the `git revert HEAD` command to undo the last commit.

```
user@host:~/project$ git revert HEAD
user@host:~/project$ git log --oneline
2181656 (HEAD -> main) Revert "Changed my_script.py"
1d47a66 Changed my_script.py
2bd7b04 Initial commit.
```



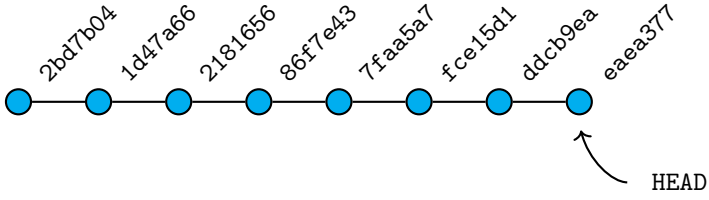
- Note that `git revert` works by creating a commit that applies the changes of previous commits in reverse. This helps maintain a contiguous log of changes made to the project.
- We can also specify a range of commits to undo. For example, `git revert HEAD~3` undoes the last 4 commits.

Recap

- So what does this mean?

- Git allows us to track changes to a project.
  - Means we can clearly track the progression of a project and see when it breaks.
  - Can revert to a previous stage to undo mistakes.
- Git has a clear and definite history.
  - A comprehensive log of what changes were made to a project and when.

- *"Commit early, commit often."*



Branching

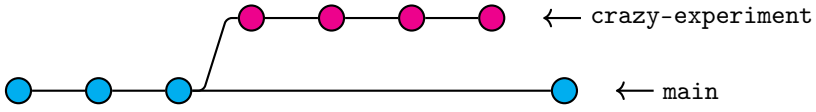
- Branches allow us to create a parallel version of our project.
- The command `git branch <name>` creates a new branch, while `git branch` lists the branches.

```
user@host:~/project$ git branch crazy-experiment
user@host:~/project$ git branch

* main
  crazy-experiment


user@host:~/project$ git checkout crazy-experiment

Switched to branch 'crazy-experiment'
```

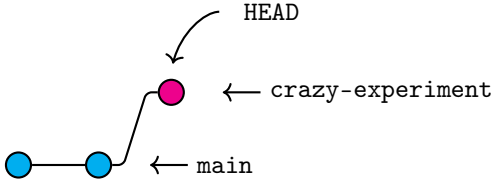


- Branching is one of the most useful features of Git.
- "Copies" the state of the project at a given commit - no more copying entire projects!

Committing to Branches

```
user@host:~/project$ git checkout crazy-experiment
user@host:~/project$ edit my_script.py
user@host:~/project$ git commit my_script.py
user@host:~/project$ git log --oneline

4e7aecd (HEAD -> crazy_experiment) Experimental changes to my_script.py
1d47a66 (main) Changed my_script.py
2bd7b04 Initial commit.
```



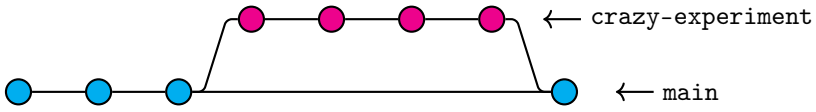

```
user@host:~/project$ git checkout main
user@host:~/project$ git log --oneline

1d47a66 (HEAD -> main) Changed my_script.py
2bd7b04 Initial commit.
```

Merging Branches

- We can then apply the changes made to one branch onto another. This is called "merging".
- First, `git checkout` the branch you want to merge *into*. Then, use the `git merge` command.

```
user@host:~/project$ git checkout main
user@host:~/project$ git merge --commit crazy-experiment
```



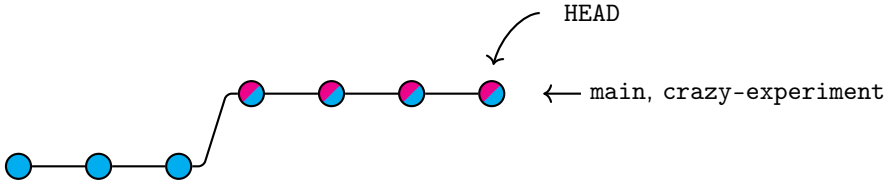
- The `--commit` flag specifies we want to create a commit with the merge result.

- **Caution:** Merging can be complicated!
  - Changes made to one branch may conflict with the changes made in another.
  - When conflicts arise, we need to specify which changes to keep.
  - Best if no changes have been made to the branch we are merging into.
  - Okay if no *overlapping* changes have been made.

Fast-Forward Merges

- If no changes have been made to the original branch, we can do a fast-forward merge.
- Applies the changes directly to the receiving branch.
- Updates the `HEAD` of the receiving branch.

```
user@host:~/project$ git checkout main
user@host:~/project$ git merge crazy-experiment
```

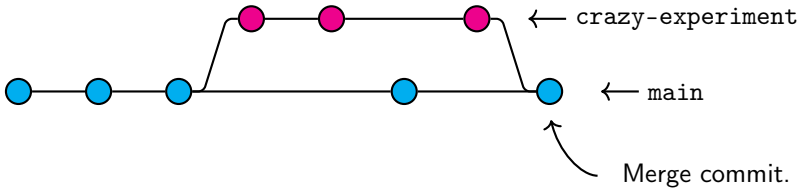


- This is the easiest way to merge branches. However, this is not always possible.
- Does not create a new commit.
- Keeps history linear.

True Merges

- If the receiving branch has diverged, we must do a true merge.
- Creates a new commit that specifies the changes.

```
user@host:~/project$ git checkout main
user@host:~/project$ git merge crazy-experiment
```

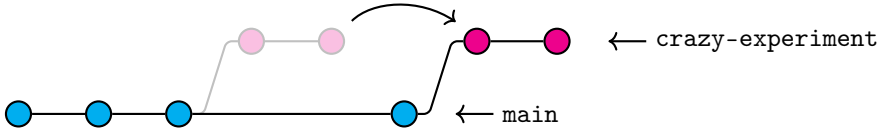


- May create “merge conflicts”.
- E.g. the same line in a file has been changed in both branches - Which one do we keep?
- Merge conflicts must be resolved before the merge commit can be completed.

Rebasing a Branch

- We can move the base of a branch using `git rebase`.

```
user@host:~/project$ git checkout crazy-experiment
user@host:~/project$ git rebase main
```

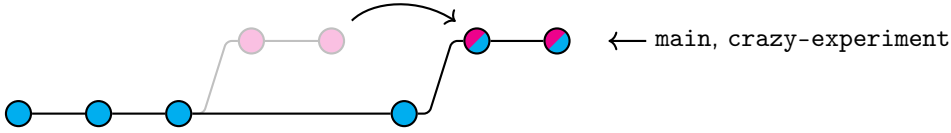


- This re-applies the branch commits on top of the new base.
- **Caution:** rebasing is still a complicated procedure.
  - May still cause merge conflicts.
  - Use `git rebase --abort` to cancel a rebase.
  - Can cause major problems with shared repositories!

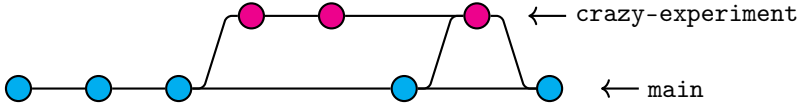
Rebase & Merge vs. Merge Twice

- It is often simpler to rebase and then perform a fast-forward merge.

```
user@host:~/project$ git checkout crazy-experiment
user@host:~/project$ git rebase main
user@host:~/project$ git checkout main
user@host:~/project$ git merge crazy-experiment
```




```
user@host:~/project$ git checkout crazy-experiment
user@host:~/project$ git merge main
user@host:~/project$ git checkout main
user@host:~/project$ git merge crazy-experiment
```

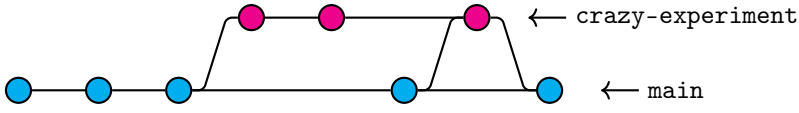


Recap

- Branching allows for concurrent versions/copies of the project files.
  - Can easily switch back and forth between branches.
  - Develop on each branch independently & merge as needed.
- Branching offers lots of flexibility in structuring & organizing code.
  - Too much flexibility can be bad, keep it simple!

- Okay, great. Now what?

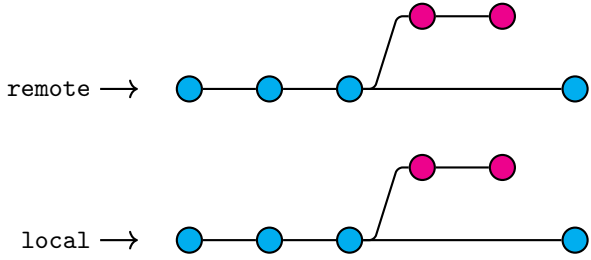
- Shared projects & repositories.
- New challenges for avoiding conflicts.



GitHub & Remote Repos

- Git allows us to create a centralized repo that can be shared among team members.
- Shared files & history of project changes - no more emailing zip files!
- GitHub is a service which hosts Git repos.

- With a remote repo, we have two copies: one on a remote server, and another kept locally.



- **Challenge:** How do we ensure that everyone is working with the latest code?

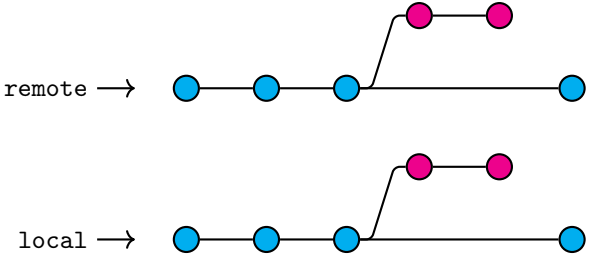
Cloning Repos

- "Cloning" a repo means downloading a local copy that we can work on.
- Say there is a remote repo on GitHub with the following files: `my_script.py` `README.md`.

```
user@host:~$ git clone https://github.com/<user>/<your_repo>.git project
user@host:~$ cd project
user@host:~/project$ ls -A

.git  my_script.py  README.md
```

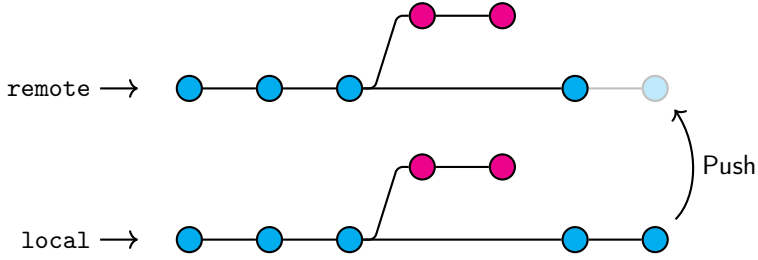
- This creates a local copy of the repo in the `project` folder.



Pulling & Pushing Changes

- "Pulling" and "pushing" refers to downloading & uploading changes to the remote repo.
- To do this, we use the `git pull` and `git push` commands.

```
user@host:~/project$ git pull
user@host:~/project$ git push
```



- **Best practice:** Pull changes from remote, resolve any merge conflicts, push changes to remote.
- You can use `git pull --rebase` to rebase your changes on top of the remote changes.
- Once your commits are pushed, they become a part of the shared history.

Uploading Existing Repos to GitHub

- Uploading existing projects to GitHub is very easy.
- Say you have a local project with the following files: `my_script.py` and `README.md`.

```
user@host:~$ cd project
user@host:~/project$ git init
user@host:~/project$ ls -A

.git  my_script.py  README.md


user@host:~/project$ git commit -a -m "Initial commit."
```

- Create an empty repo on GitHub, e.g. `https://github.com/<user>/<your_repo>.git`.

```
user@host:~/project$ git remote add origin https://github.com/<user>/<your_repo>.git
user@host:~/project$ git push -u origin main
```

- Also serves as a great way to back up your project in case of catastrophe.
- Now, anyone who has access can clone your repo using `git clone`.

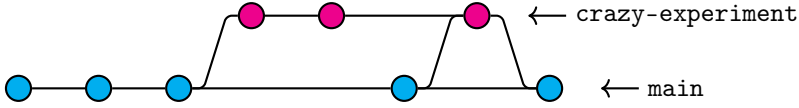
```
user@host:~$ git clone https://github.com/<user>/<your_repo>.git project
```

Avoiding Problems With Remote Repos

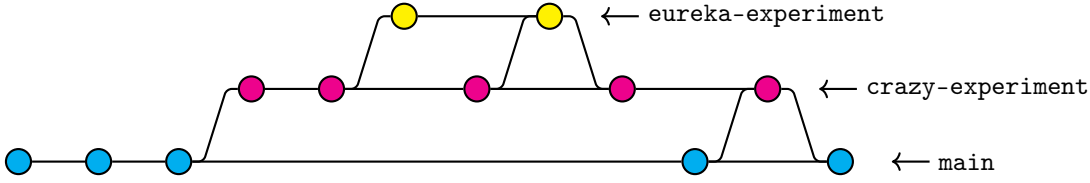
- Changes to the Git history are not always easy to synchronize.
- Increased chance of merge conflicts.

Best Practices

- Do not rebase remote branches. Instead, do two merges.



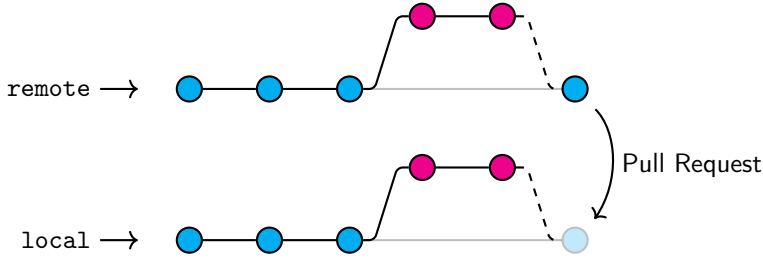
- Branches are cheap. Use short-lived branches and merges liberally.
  - Create a temporary branch, make changes, merge, and then delete the branch.



- Commit, pull, and push often. Ensures everyone is working with the latest changes.
- Use `git revert` instead of `git reset`.

Pull Requests

- A "pull request" (PR) allows for code review & discussion before merges.
- A push forces your changes onto the remote. A pull request asks the remote to pull your changes.
- Means you can control the modifications to the original code.



- Pull requests can be checked out, just like branches[2].

```
user@host:~/project$ git fetch origin pull/<ID>/head:<branch_name>
user@host:~/project$ git checkout <branch_name>
```

[2]Note that this creates a local *copy* of the PR into a new branch.

Advanced Topics

- GitHub introduces several other advanced features.
  - GitHub Actions - Automated code testing & execution.
  - Forks - independent copies of remote repos.
  - Issues - tracks bugs & helps resolve them.

  See `https://docs.github.com`.
- GitHub Desktop - a great graphical tool for managing Git repos.

- Git is very flexible.
- Git workflows help create standard practices among team members.
  - Gitflow.
  - Forking workflows.
  - Central branch workflow.

  See `https://www.atlassian.com/git/tutorials/comparing-workflows`

- Git submodules.
- Advanced `git checkout` and `git fetch` capabilities.
- Stashing changes.

General Recommended Workflow

- Treat the `main` branch as sacred.
- Create a parallel `development` branch off of `main`.
  - Branch off of `development` for new features.
  - Only allow pull requests from `development` to `main`.
- Keep short-lived branches local.
  - Rebase local branches off `development`.