

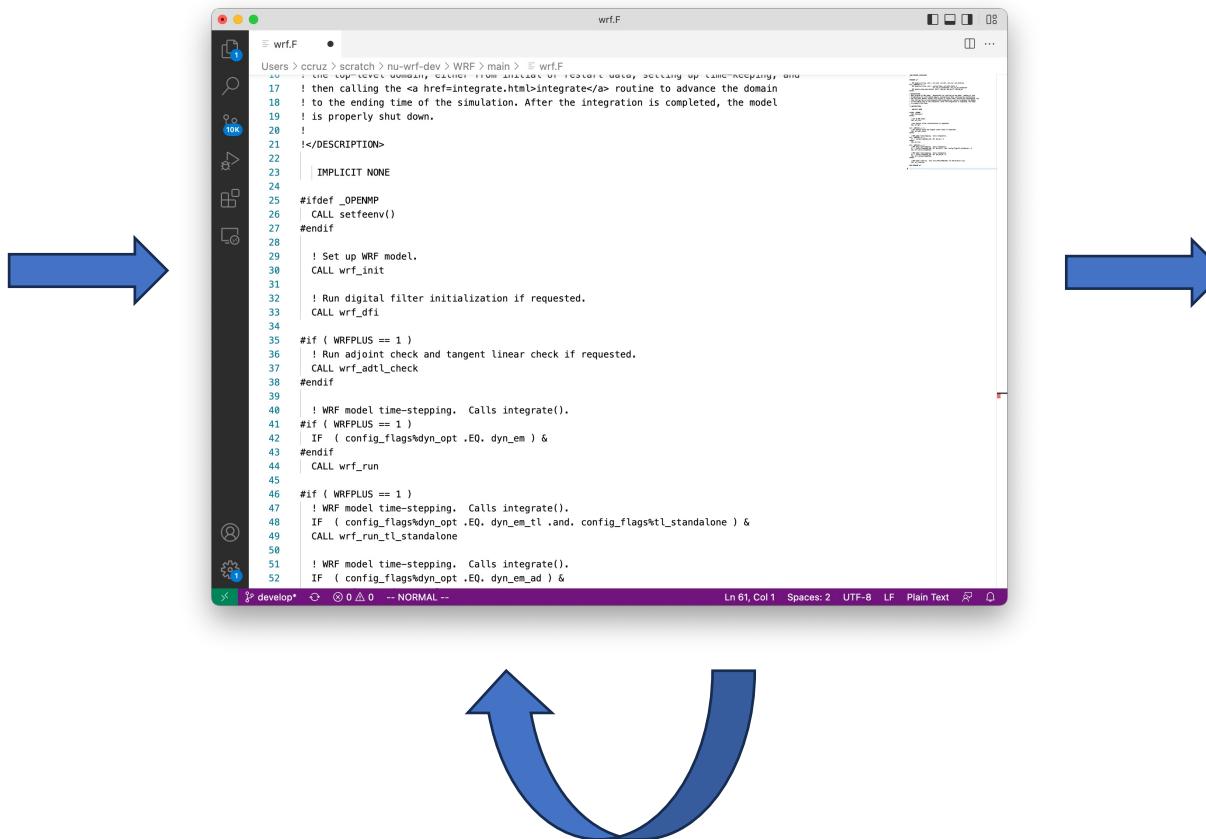
Introduction to Version Control with Git

Summer 2024 NASA Bootcamp

Motivation

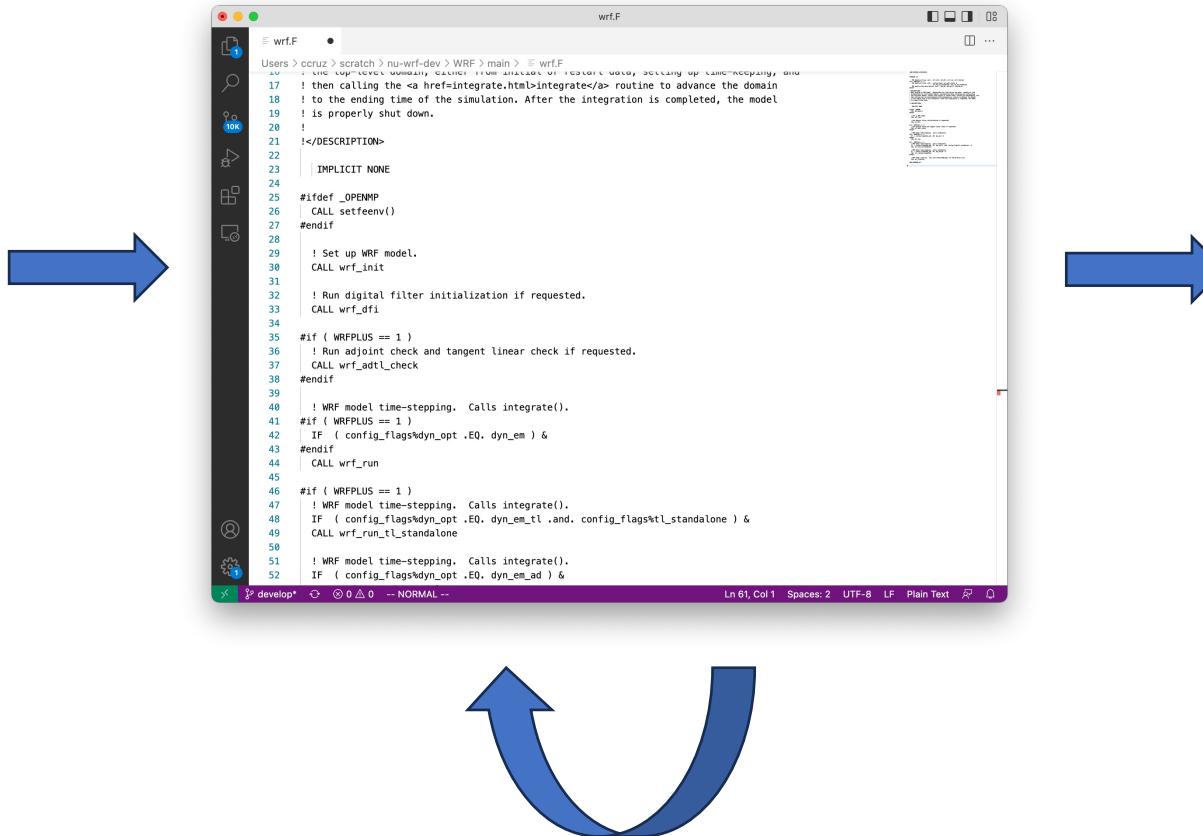
A screenshot of a terminal window titled "wrf.F". The window displays a large amount of Fortran code for the WRF model. The code includes comments explaining the purpose of various sections, such as setting up the domain, calling the "integrate" routine, and setting up the WRF model. The code is organized into sections like "<DESCRIPTION>" and "#IMPLICIT NONE". The terminal interface shows standard file navigation commands on the left and status information at the bottom.

Software development includes the continuous process of modifying programs

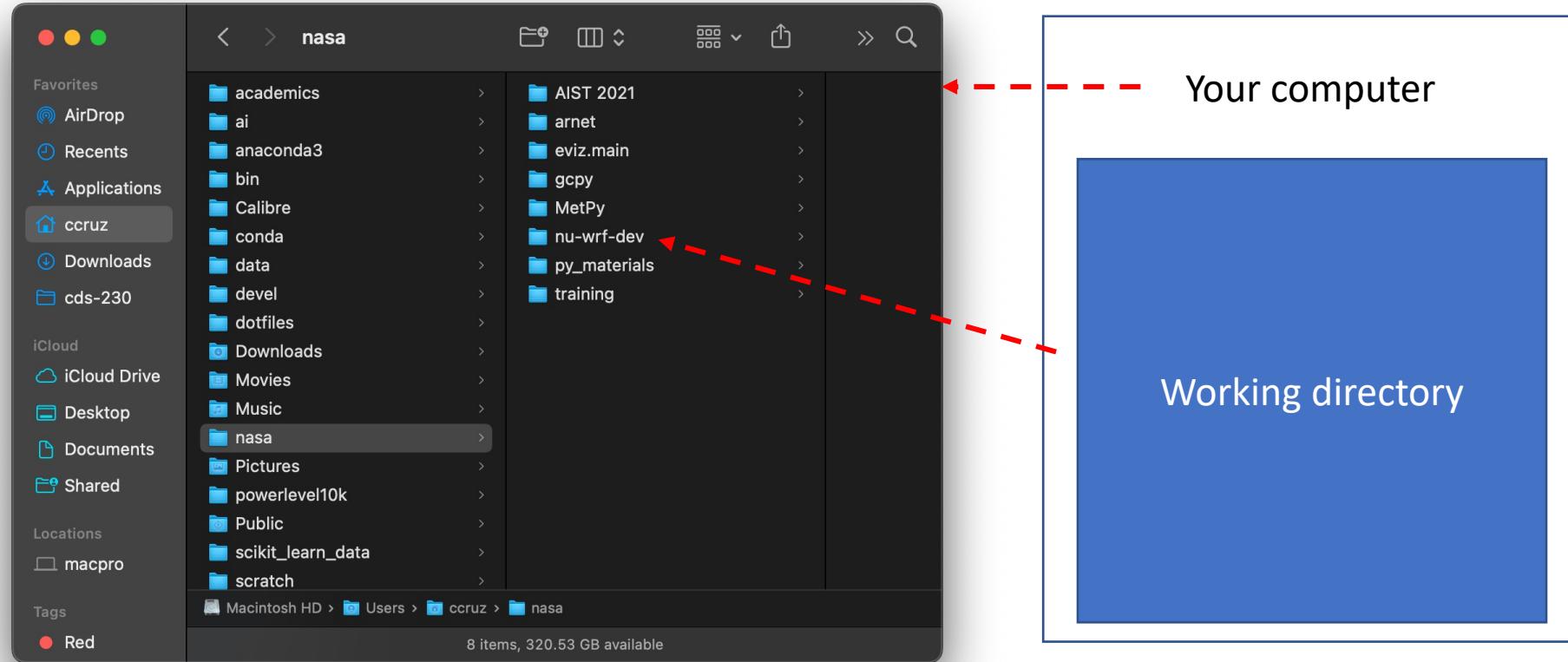


Software development includes the continuous process of modifying programs

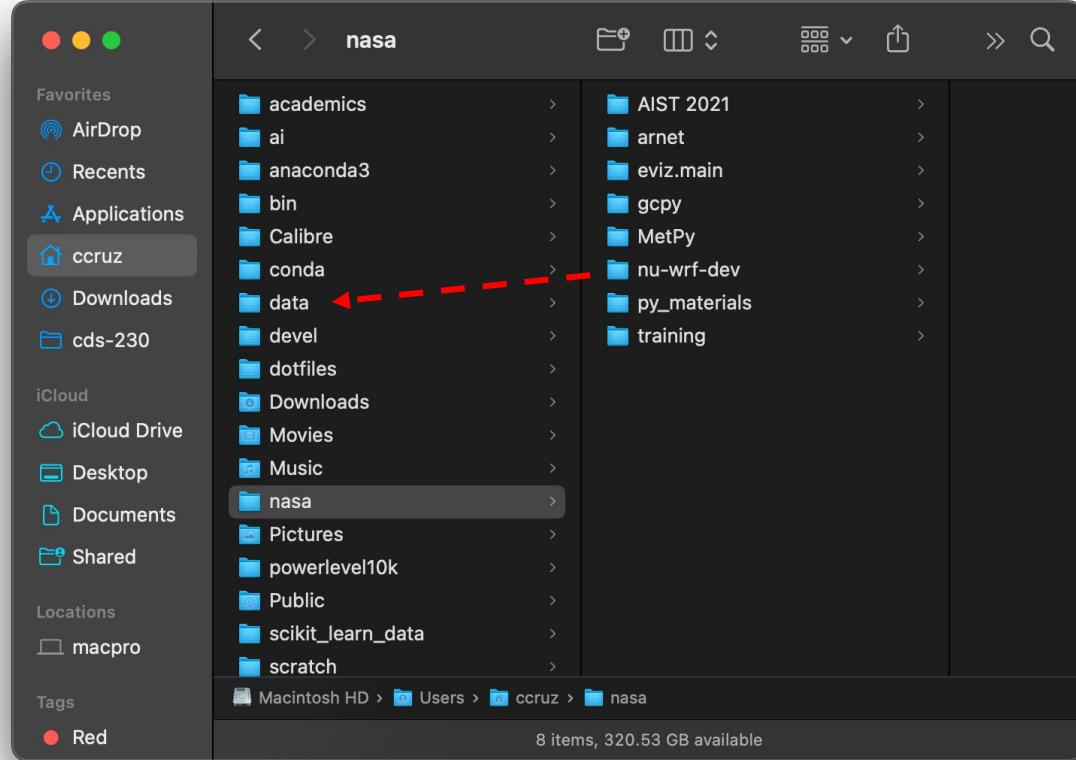
Proper code management is essential for effective and sustainable software development



Why do we need to manage source code changes?



How can we manage source code changes?

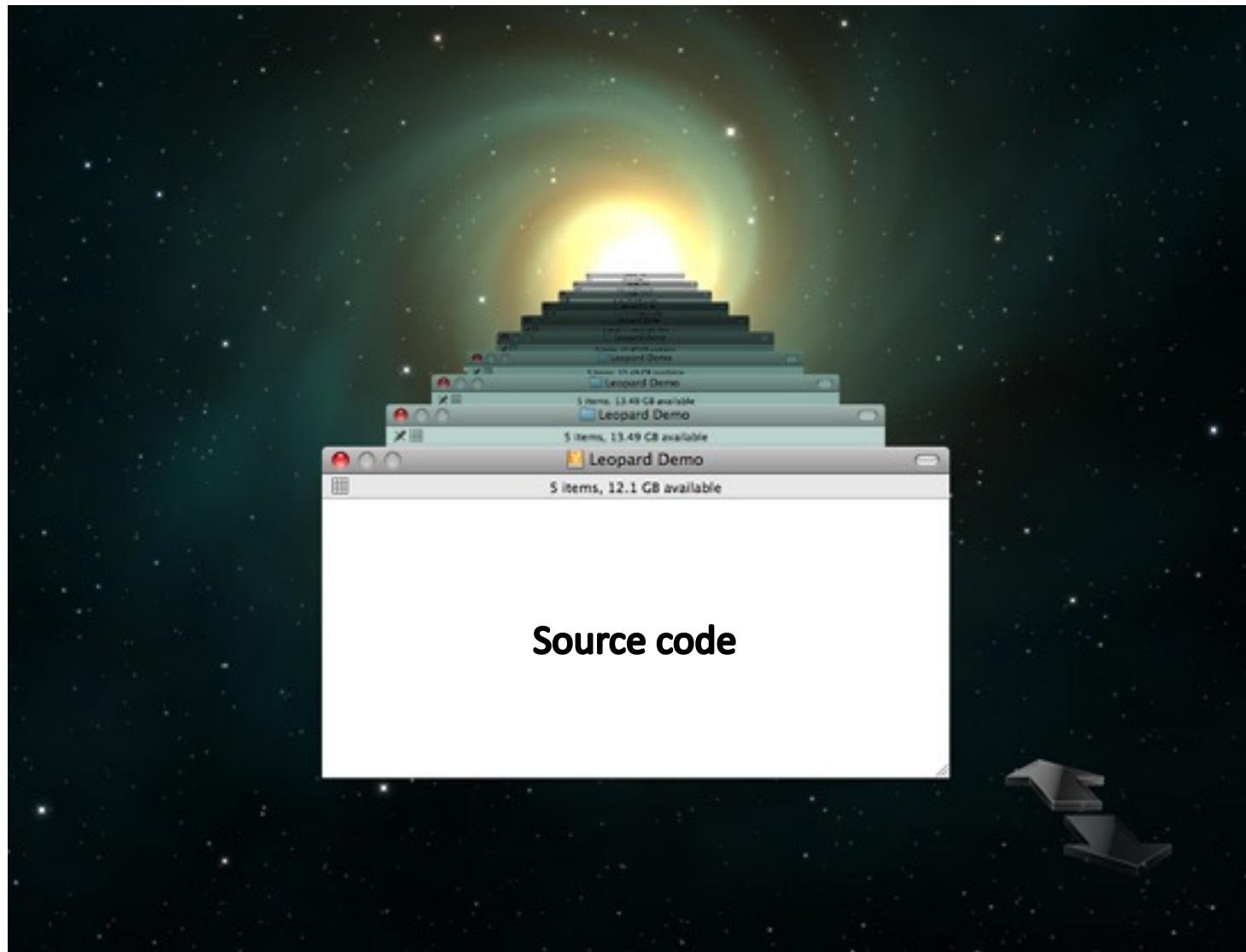


Your computer

```
[ccruz: data] $  
script.py.v4  
script.py.v3  
script.py.v2  
script.py.v1  
version_notes
```

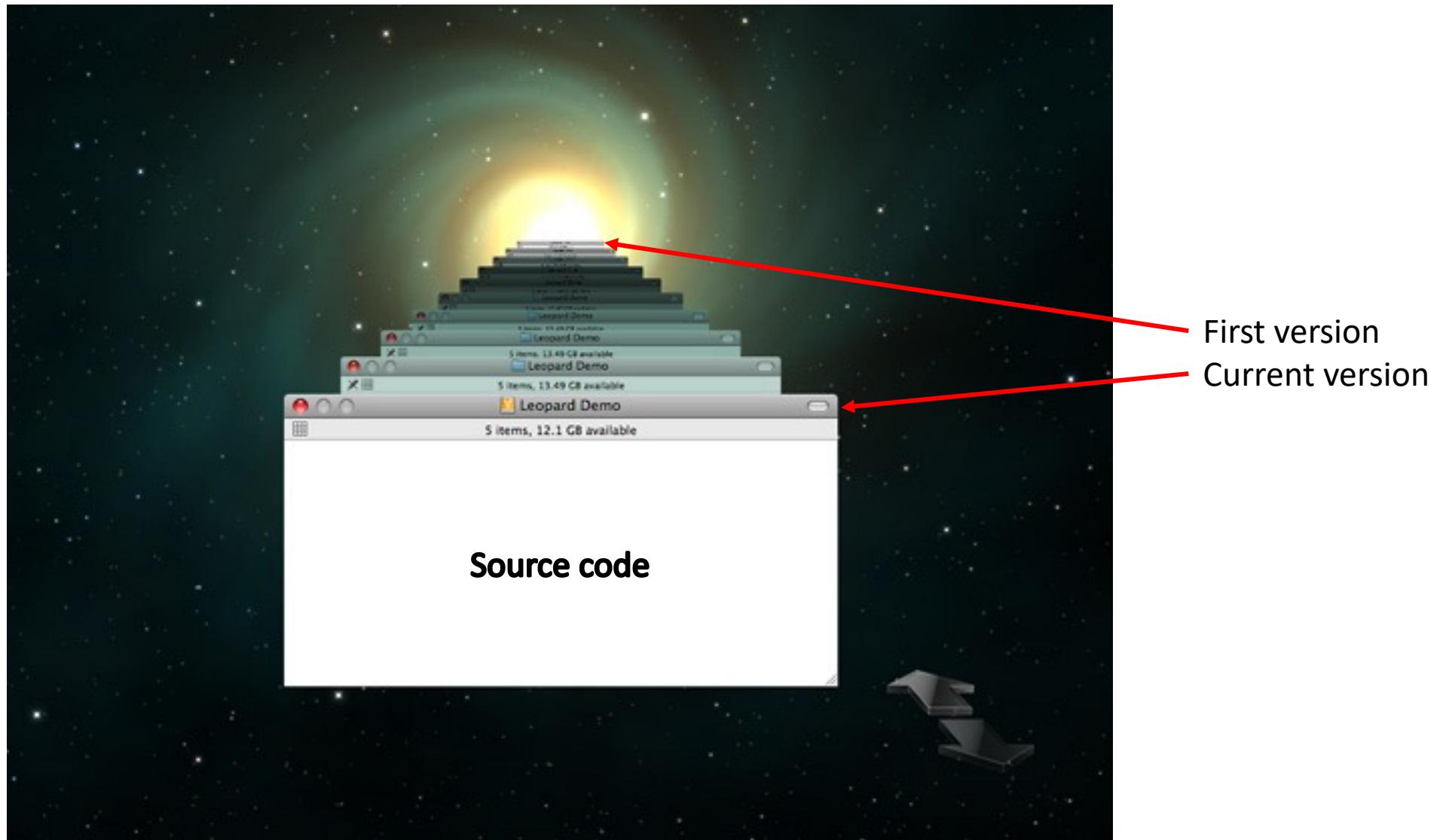
A personal “version control system”

Version Control System (VCS)

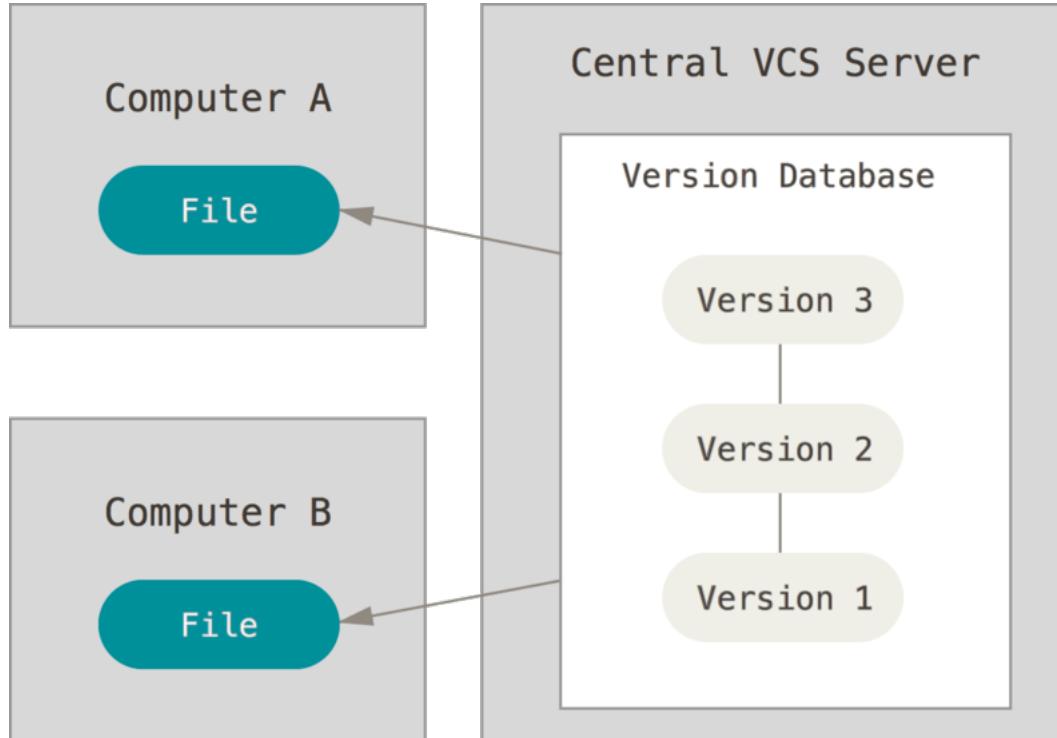


Source code

Version Control System (VCS)



Centralized VCS



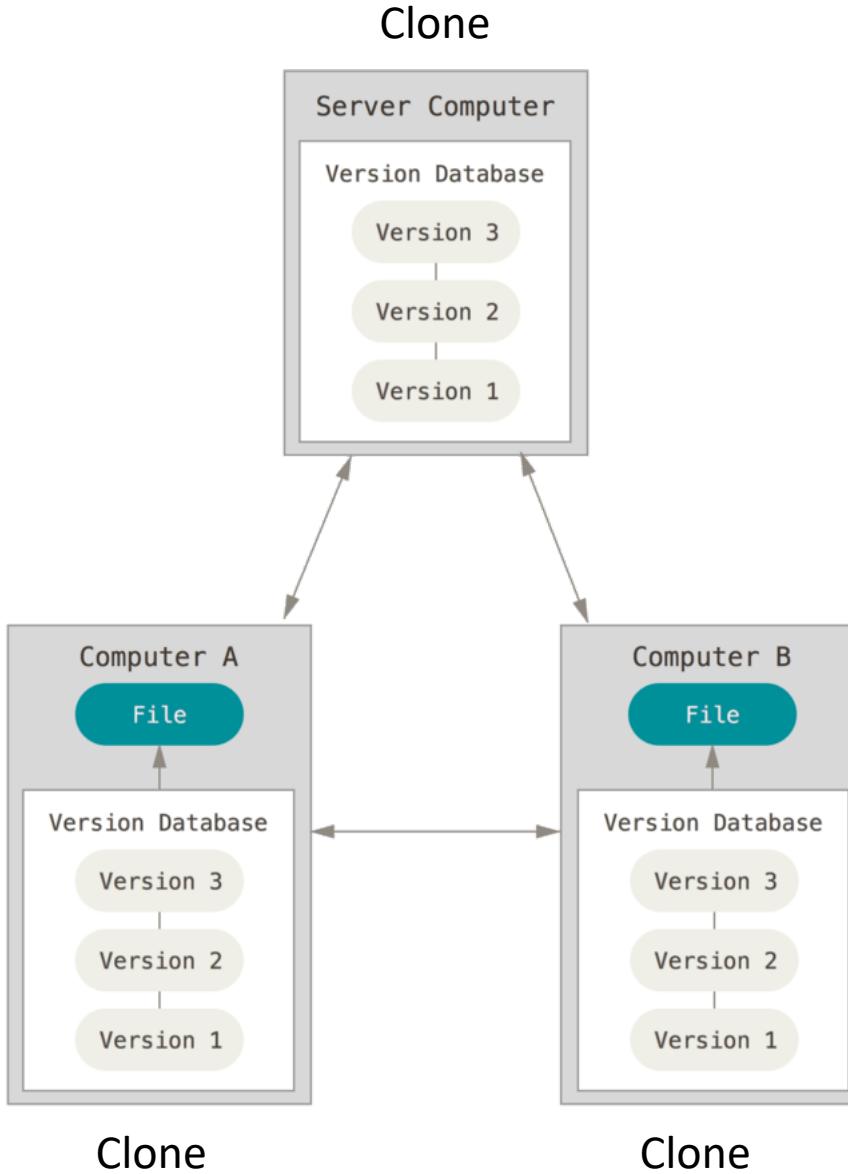
Examples:

RCS (c. 1982)

CVS (c. 1990)

Subversion (c. 2000)

Distributed VCS



Examples:

Bitkeeper (c. 2000)
Mercurial (c. 2005)
Git (c. 2005)

Git is an implementation of a distributed VCS

The screenshot shows the official Git website (<https://git-scm.com/>). At the top, the Git logo and the tagline "local-branching-on-the-cheap" are displayed. A search bar is located in the top right corner. Below the header, there are two main sections of text. The first section describes Git as a "free and open source distributed version control system" designed for efficiency. The second section highlights its ease of learning, fast performance, and unique features like "cheap local branching". To the right of the text is a diagram illustrating the distributed nature of Git. It shows seven white rectangular boxes representing repositories, connected by a network of colored arrows (red, blue, green, yellow) forming a mesh-like structure. Below the text and diagram, there are six circular icons with corresponding labels: "About" (gear icon), "Documentation" (book icon), "Downloads" (download arrow icon), "Community" (speech bubble icon), and two additional icons whose descriptions are not fully visible. On the right side of the page, there's a large image of a computer monitor displaying the latest source release information (version 2.41.0, release notes from 2023-06-01, download button for Mac). Below the monitor are links for "Mac GUIs", "Tarballs", "Windows Build", and "Source Code". At the bottom left, there's a small image of the "Pro Git" book cover. The footer contains the text: "Pro Git by Scott Chacon and Ben Straub is available to read online for free. Dead tree versions are available on [Amazon.com](#)".

git --local-branching-on-the-cheap

Search entire site...

Git is a **free and open source** distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is **easy to learn** and has a **tiny footprint with lightning fast performance**. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like **cheap local branching**, convenient staging areas, and **multiple workflows**.

About
The advantages of Git compared to other source control systems.

Documentation
Command reference pages, Pro Git book content, videos and other material.

Downloads
GUI clients and binary releases for all major platforms.

Community
Get involved! Bug reporting, mailing list, chat, development and more.

Latest source Release
2.41.0
[Release Notes \(2023-06-01\)](#)
[Download for Mac](#)

Mac GUIs Tarballs

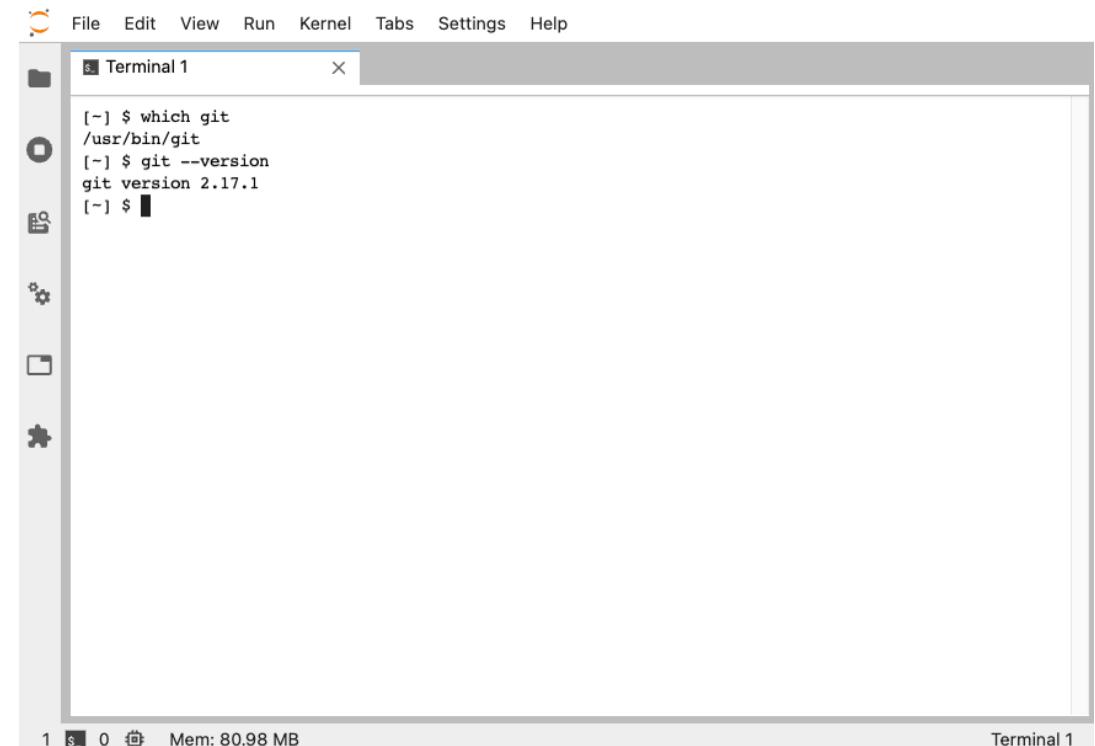
Windows Build Source Code

Pro Git by Scott Chacon and Ben Straub is available to read online for free. Dead tree versions are available on [Amazon.com](#).

Official website:
<https://git-scm.com/>

Using Git

- Git is a software tool that needs to be installed on your computer.
- You do not need to install it today.
 - You will use the SMCE platform which has Git installed



A screenshot of a terminal window titled "Terminal 1". The window has a menu bar with File, Edit, View, Run, Kernel, Tabs, Settings, and Help. The main area shows the command line output:

```
[~] $ which git  
/usr/bin/git  
[~] $ git --version  
git version 2.17.1  
[~] $
```

The terminal window is part of a larger interface with a sidebar containing icons for file, terminal, and other functions. The status bar at the bottom shows "1 \$ 0 Mem: 80.98 MB" and "Terminal 1".

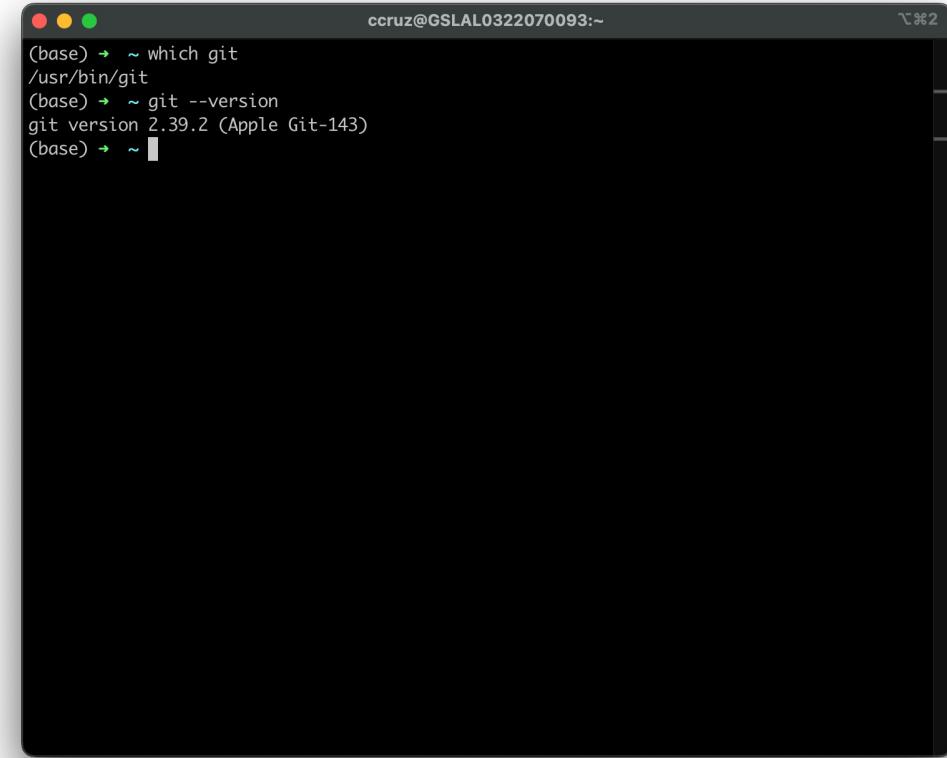
SMCE training platform: <https://training.astg.smce.nasa.gov/hub/login>

Using Git

- You may use your laptop, if it has already Git installed
- If you need to install it, you can do so later (start) here



Official website:
<https://git-scm.com/>



```
ccruz@GSLAL0322070093:~  
(base) ~ which git  
/usr/bin/git  
(base) ~ git --version  
git version 2.39.2 (Apple Git-143)  
(base) ~ |
```

A screenshot of a macOS terminal window titled "ccruz@GSLAL0322070093:~". The window shows a command-line session where the user runs "which git" to find the path to the git executable, and then runs "git --version" to check the installed version, which is 2.39.2 (Apple Git-143). The terminal has a dark background with light-colored text.

Creating and modifying source code

Code Editors: vim, emacs, nano, etc.

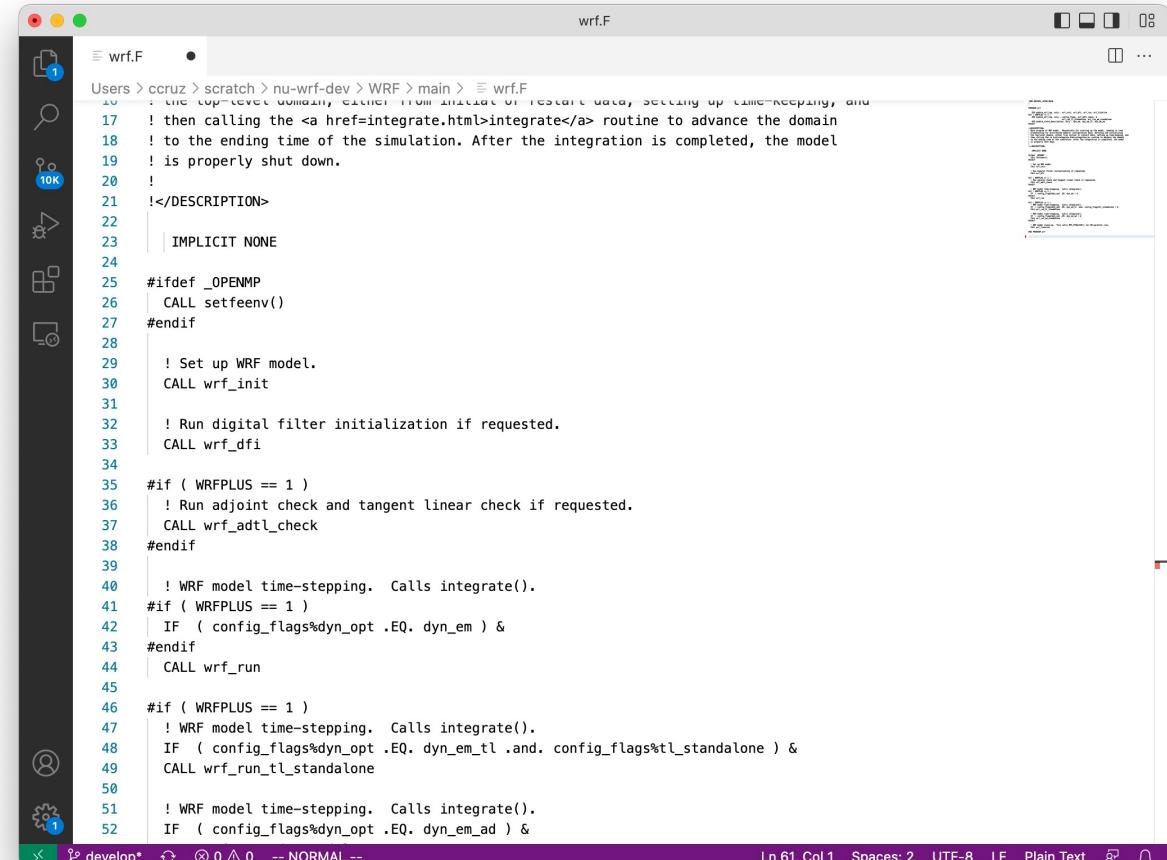
IDEs: VS Code, PyCharm, Sublime*, etc.



vi wrf.F

```
16 ! the top-level domain, either from initial or restart data, setting up time-keeping, and
17 ! then calling the <a href=integrate.html>integrate</a> routine to advance the domain
18 ! to the ending time of the simulation. After the integration is completed, the model
19 ! is properly shut down.
20 !
21 !</DESCRIPTION>
22
23 IMPLICIT NONE
24
25 #ifdef _OPENMP
26 CALL setfeenv()
27 #endif
28
29 ! Set up WRF model.
30 CALL wrf_init
31
32 ! Run digital filter initialization if requested.
33 CALL wrf_dfi
34
35 #if ( WRFPLUS == 1 )
36 ! Run adjoint check and tangent linear check if requested.
37 CALL wrf_adtl_check
38 #endif
39
40 ! WRF model time-stepping. Calls integrate().
41 #if ( WRFPLUS == 1 )
42 IF ( config_flags%dyn_opt .EQ. dyn_em ) &
43 #endif
44 CALL wrf_run
45
46 #if ( WRFPLUS == 1 )
47 ! WRF model time-stepping. Calls integrate().
48 IF ( config_flags%dyn_opt .EQ. dyn_em_tl .and. config_flags%tl_standalone ) &
49 CALL wrf_run_tl_standalone
```

vim



wrf.F

```
16 ! the top-level domain, either from initial or restart data, setting up time-keeping, and
17 ! then calling the <a href=integrate.html>integrate</a> routine to advance the domain
18 ! to the ending time of the simulation. After the integration is completed, the model
19 ! is properly shut down.
20 !
21 !</DESCRIPTION>
22
23 IMPLICIT NONE
24
25 #ifdef _OPENMP
26 CALL setfeenv()
27 #endif
28
29 ! Set up WRF model.
30 CALL wrf_init
31
32 ! Run digital filter initialization if requested.
33 CALL wrf_dfi
34
35 #if ( WRFPLUS == 1 )
36 ! Run adjoint check and tangent linear check if requested.
37 CALL wrf_adtl_check
38 #endif
39
40 ! WRF model time-stepping. Calls integrate().
41 #if ( WRFPLUS == 1 )
42 IF ( config_flags%dyn_opt .EQ. dyn_em ) &
43 #endif
44 CALL wrf_run
45
46 #if ( WRFPLUS == 1 )
47 ! WRF model time-stepping. Calls integrate().
48 IF ( config_flags%dyn_opt .EQ. dyn_em_tl .and. config_flags%tl_standalone ) &
49 CALL wrf_run_tl_standalone
50
51 ! WRF model time-stepping. Calls integrate().
52 IF ( config_flags%dyn_opt .EQ. dyn_em_ad ) &
```

Visual Studio

Configuring Git



A terminal window titled "ccruz@GSLAL0322070093:~". The window shows the following command history:

```
(base) ~ which git  
/usr/bin/git  
(base) ~ git --version  
git version 2.39.2 (Apple Git-143)  
(base) ~ █
```

The text "git config --help" is displayed in white at the bottom left of the terminal window.

- **System:** /etc/.gitconfig
- **User:** \$HOME/.gitconfig ←
- **Project:** my_project/.git/config

Git commands to edit the configuration:

git config --system [options] (system)
git config --global [options] (user) ←
git config [options] (project)

Exercise

Run the following *git config* commands on your terminal:

\$ git config --global user.name "Your Name Goes Here"

Sets the name you want attached to your commit transactions

\$ git config --global user.email "yourusername@domain.com"

Sets the email you want to be attached to your commit transactions

\$ git config --global core.editor vim ← *Specify editor here, e.g. vim, emacs, nano, pico*

Sets default editor

\$ git config --global init.defaultBranch main ← *More about branches later*

Sets default branch

This will create a file named \$HOME/.gitconfig with the following contents:

[user]

name = Your Name Goes Here

Check your settings by running:

email = yourusername@domain.com

\$ git config --list

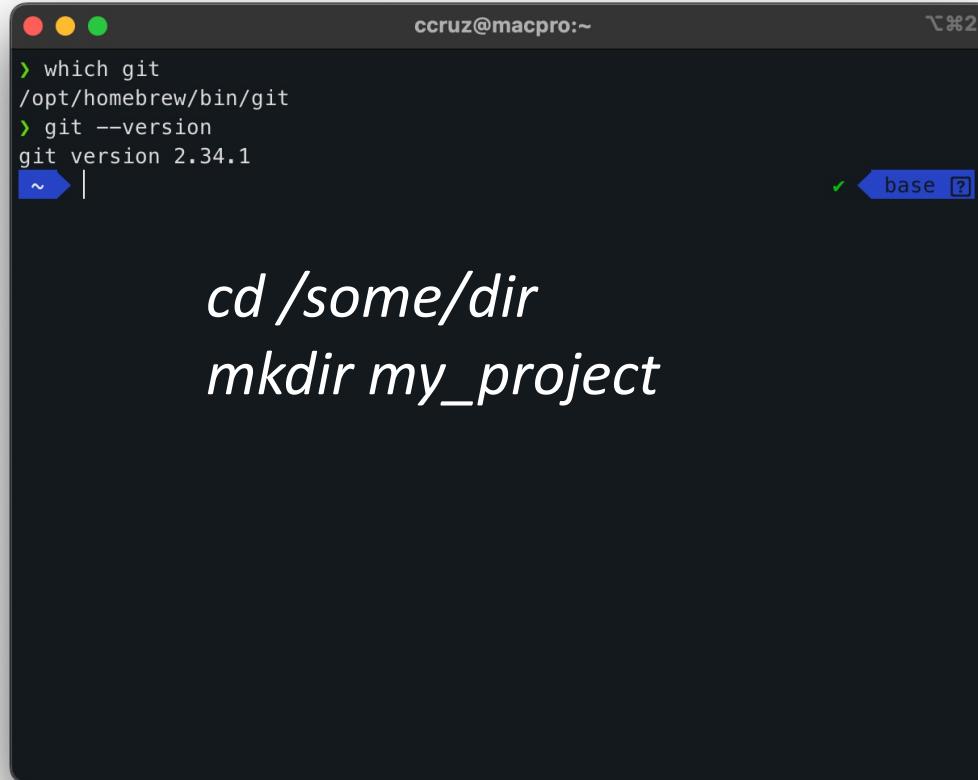
[core]

editor = /usr/bin/vim

Create a Working Directory*

Exercise

run the following commands on a terminal:



```
ccruz@macpro:~
```

```
> which git
/opt/homebrew/bin/git
> git --version
git version 2.34.1
~ |
```

cd /some/dir

mkdir my_project

For this tutorial, go to some directory on your computer and create a working directory called *my_project*.

*Note that in practice, the working directory will generally not be empty.

Creating a “repo”

\$ git init



```
[ccruz:my_project] $ git init
Initialized empty Git repository in /Users/ccruz/scratch/my_project/.git/
[ccruz:my_project] $
```

Creating a “repo”

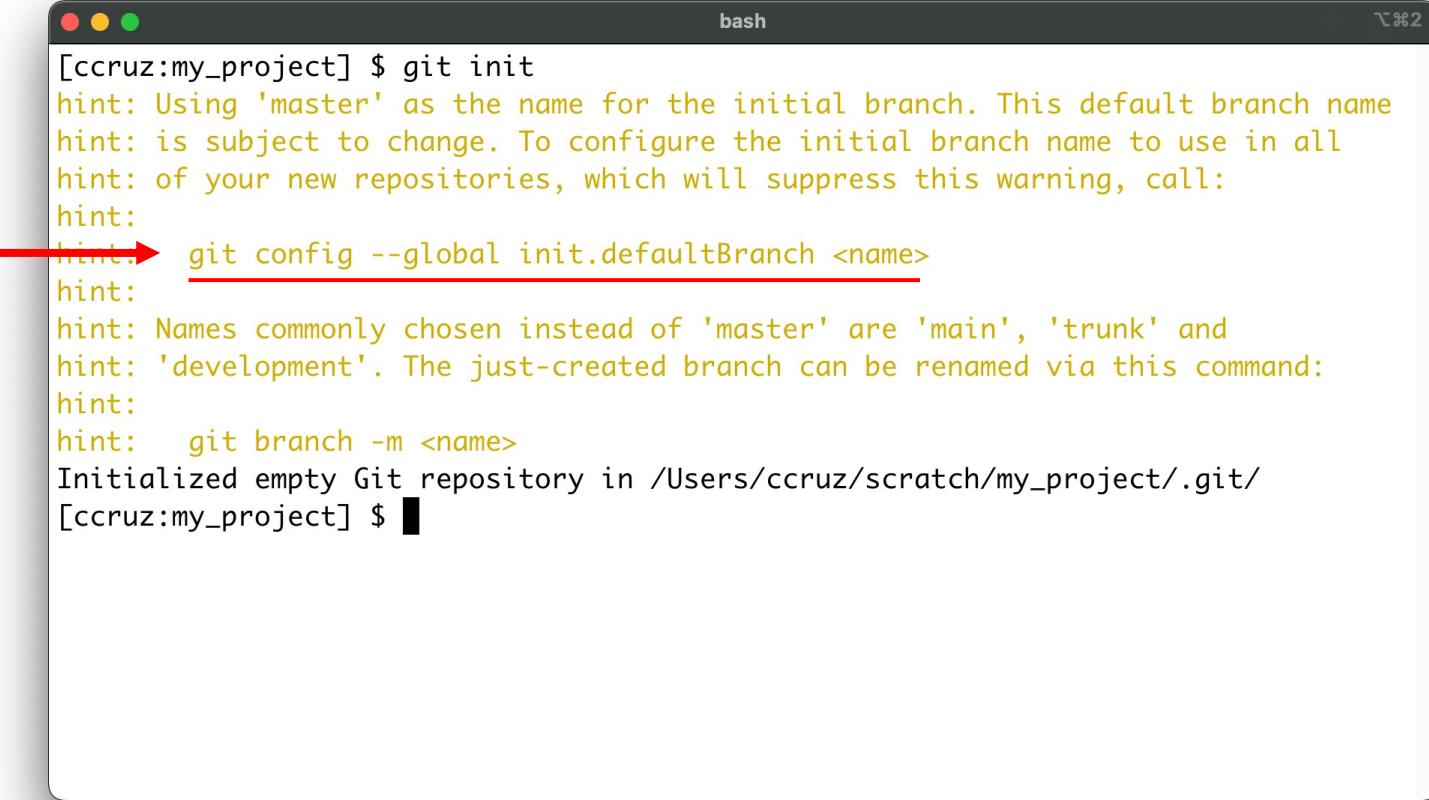
\$ git clone



```
[ccruz:scratch] $ git clone https://github.com/NASA-LIS/LISF
Cloning into 'LISF'...
remote: Enumerating objects: 42484, done.
remote: Counting objects: 100% (42484/42484), done.
remote: Compressing objects: 100% (10299/10299), done.
remote: Total 42484 (delta 31823), reused 42107 (delta 31691), pack-reused 0
Receiving objects: 100% (42484/42484), 238.61 MiB / 32.76 MiB/s, done.
Resolving deltas: 100% (31823/31823), done.
Updating files: 100% (7057/7057), done.
[ccruz:scratch] $
```

Creating a “repo”

\$ *git init*

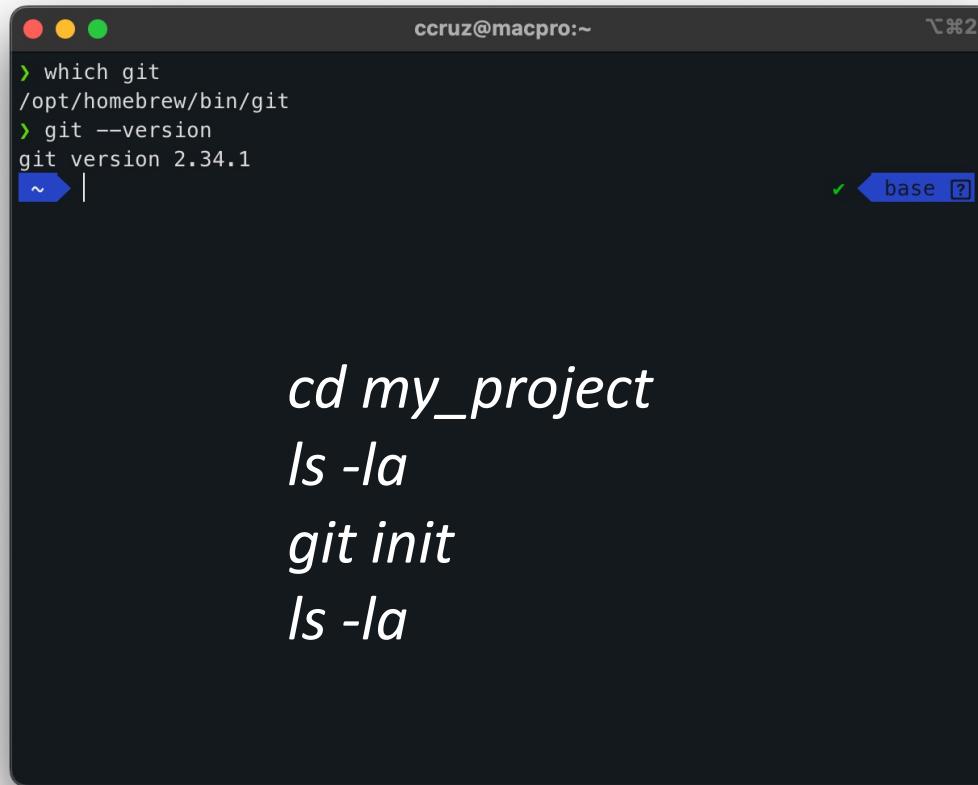


```
[ccruz:my_project] $ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint: → git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/ccruz/scratch/my_project/.git/
[ccruz:my_project] $
```

Create an Empty Repository

Exercise

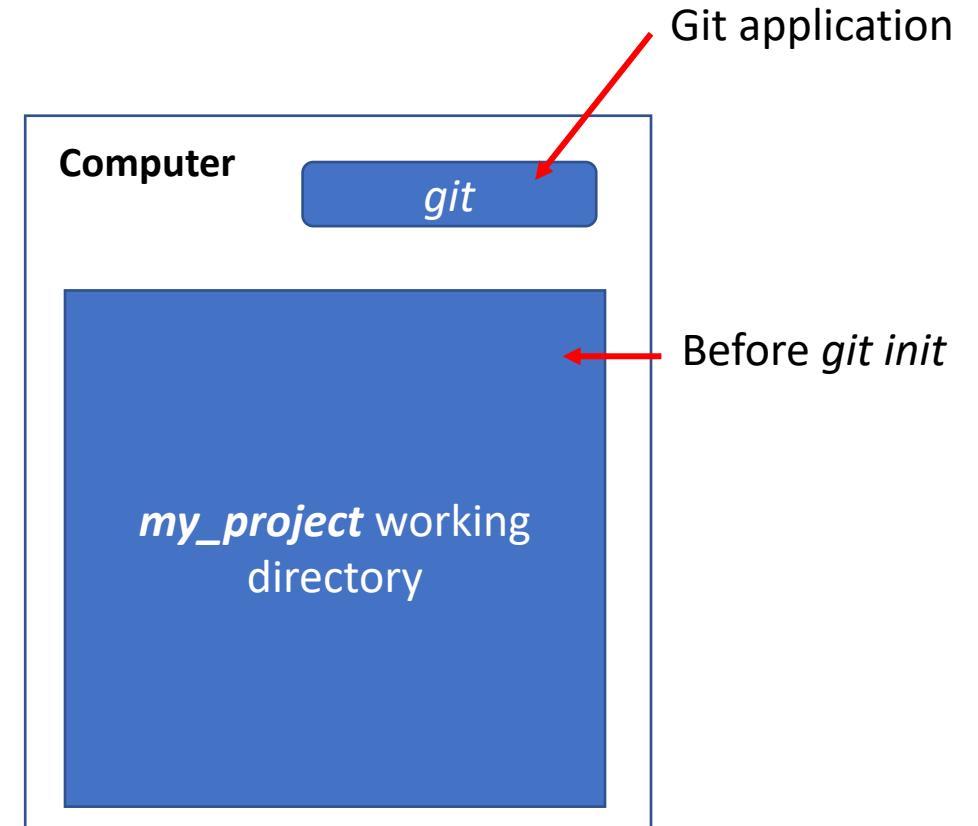
run the following commands on your terminal:



```
ccruz@macpro:~
```

```
> which git  
/opt/homebrew/bin/git  
> git --version  
git version 2.34.1  
~ |
```

cd my_project
ls -la
git init
ls -la



Create an Empty Repository

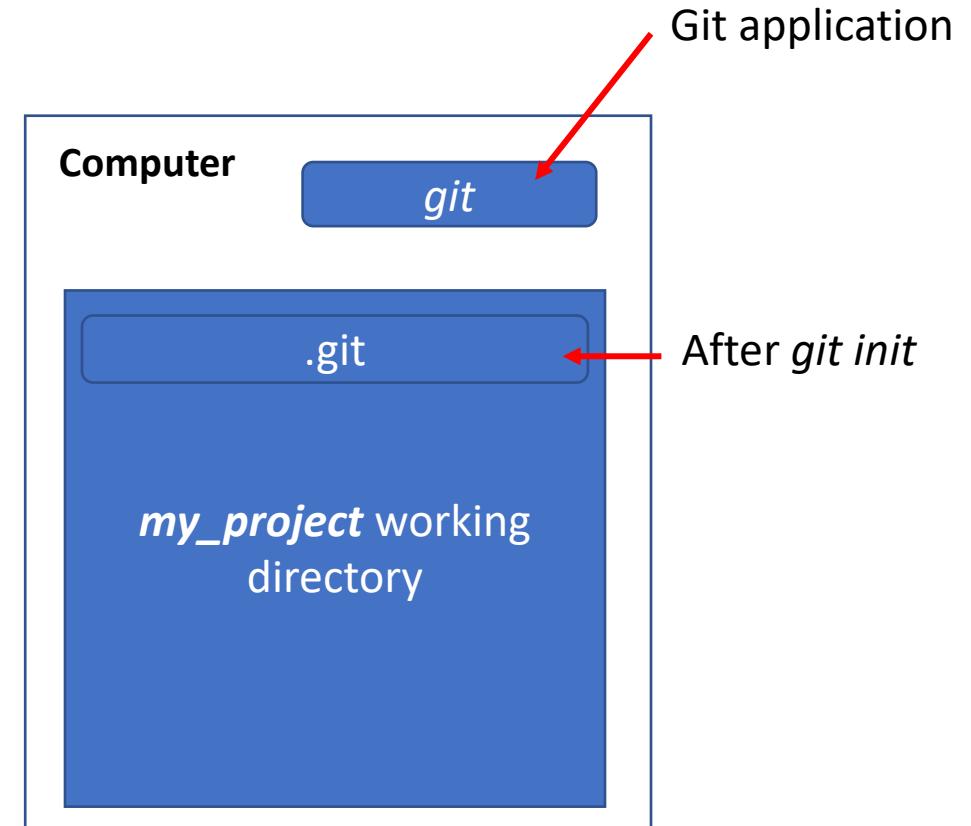
Exercise

run the following commands on your terminal:

```
ccruz@macpro:~
```

```
> which git  
/opt/homebrew/bin/git  
> git --version  
git version 2.34.1  
~ |
```

cd my_project
ls -la
git init
ls -la



init: creates a Git repository called *.git*

Basic commands

All commands start with *git*

Create repositories

\$ git init [project-name]

Creates a new local repository with the specified name



\$ git config

Configure Git's look and operation



Examine a repository

\$ git status

Lists all new or modified files to be committed

\$ git log

Show commit history

Save changes

\$ git add [file]

Snapshots the file in preparation for versioning

\$ git commit -m "[descriptive message]"

Records file snapshots permanently in version history

\$ git diff

Shows file differences not yet **staged**

\$ git stash

Shelves changes

Undo changes

\$ git reset [file]

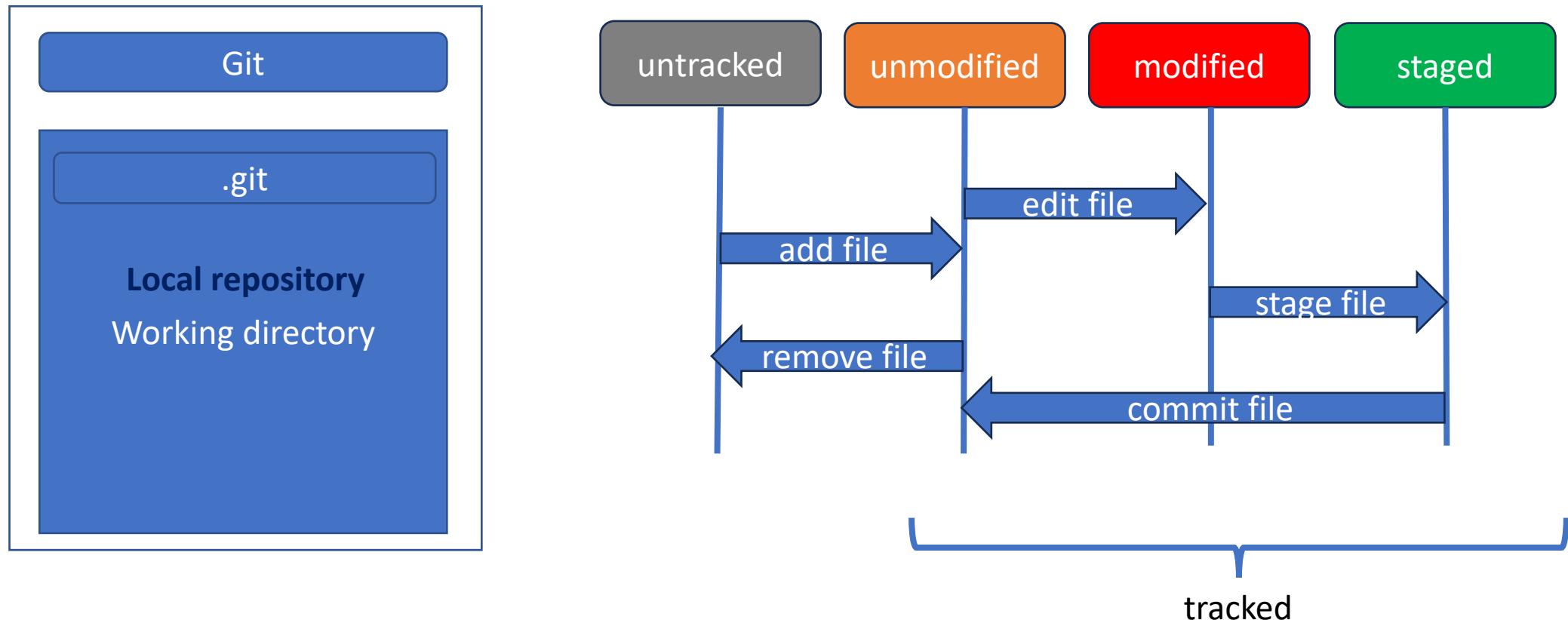
Unstages the file, but preserve its contents

\$ git restore [file]

Restore file to last committed version

Tracking Files

Files not stored in the Git repo, that is files unknown to Git, are said to be *untracked*. Otherwise, they are *tracked* or *ignored*.



Create a file

Exercise

- Open a **Terminal** and in the ***my_project*** directory you just created, create a file named *hello.py* as follows (or use your favorite editor):

```
echo "print('Hello')" > hello.py
```

- Verify its contents by running

```
ls  
cat hello.py
```

Saving Files

Exercise

Go into your local repo and check its status

git status

Add and commit the file created earlier

git add hello.py

git status

git commit -m "Add new file" hello.py

Check repo status again

git status

Let's check the history

git log

} *add and commit* transactions are associated with unique checksums called SHA1 values

Editing Files

Exercise

Edit `hello.py` so that it reads:

```
print('Hello, world!')
```

Save. Now run

```
git status
```

```
git diff hello.py ← What do you see?
```

Add and commit file

Let's check the history. Run

```
git log
```

Git Aliases

Exercise

Run the following *git config* commands on your terminal:

```
git config --global alias.ci "commit"
```

```
git config --global alias.st "status"
```

```
git config --global alias.slog "log --oneline --topo-order --graph"
```

```
git config --global alias.co "checkout"
```

This will create a section named [alias] in the \$HOME/.gitconfig file:

[alias]

ci = commit

etc...

.gitignore

- Files in Git can be tracked, untracked, or ignored.
- **Ignored files are usually machine-generated files that can be derived from your repository source or should otherwise not be committed.** For example:
 - Python-generated files *.pyc*
 - Python-generated directories *__pycache__*
 - Fortran/C intermediate files *.o .exe*
 - Mac hidden files *.DS_Store*
 - IDE-generated directories such as *.idea*
 - Etc.
- You can track these files, and ignore them, in a special file named *.gitignore*.

Create a `.gitignore` file

Exercise

Create a `.gitignore` file in your working directory. Its contents should be:

```
*.pyc  
__pycache__  
*.log
```

Add and commit the `.gitignore` file to your repo.

Deleting and moving files; saving state

- To remove a file from the working tree

git rm filename

- To move or rename a file or directory

git mv filename target

- To shelve changes that you have made to your working directory so that you can do other work, and then come back later and re-apply them:

git stash

git stash apply or *git stash pop*

Cancelling operations

- To unstage a filename

git reset filename

- To unmodify an unstaged file

git checkout filename

or

git restore filename

Git operations

Exercise

Create a TODO file with some content. Add it to the staging area as follows

```
git add TODO  
git status
```

Remove TODO from the staging area and run *git status* again <<< Do not run *rm TODO*

```
git reset TODO  
git status
```

Now create a Python script called *power.py* with the following code:

```
def power(x):  
    return x**2
```

Add and commit. Run *git log* to check your repo history

Git operations

Exercise

Edit *power.py* by adding a comment at the top, e.g.:

```
# power function
def power(x):
    return x**2
```

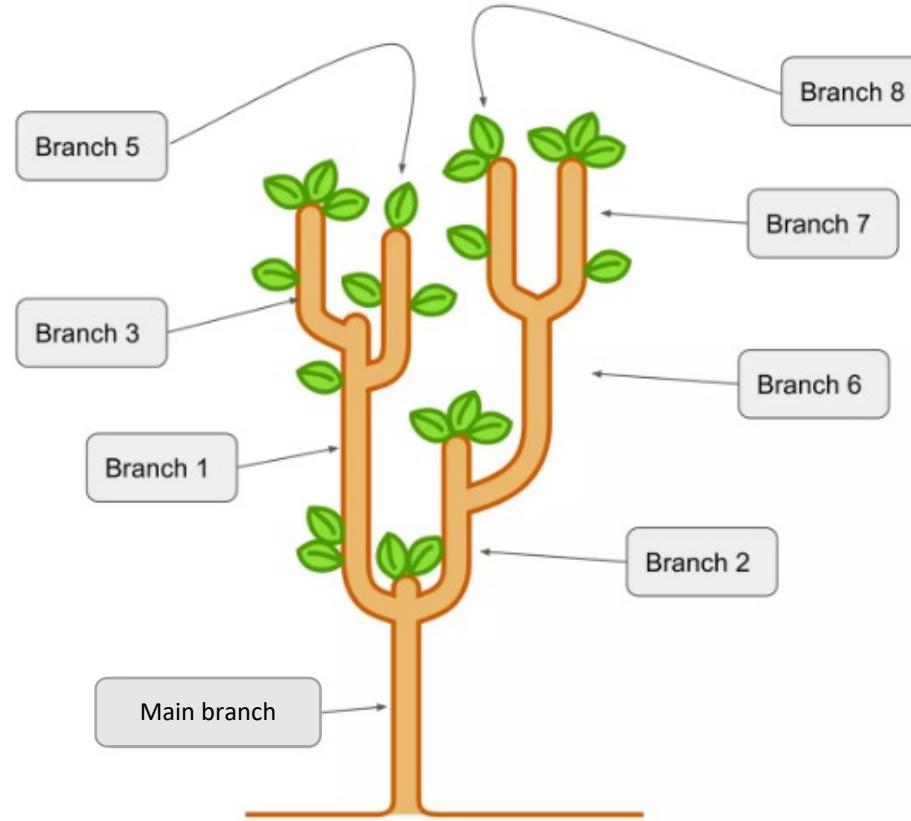
Run the following commands and observe what happens

```
git status
git diff power.py
git checkout power.py
git status
git diff power.py
```

Optional exercise: use *git mv* to rename *power.py* to *square.py*. Add and commit.

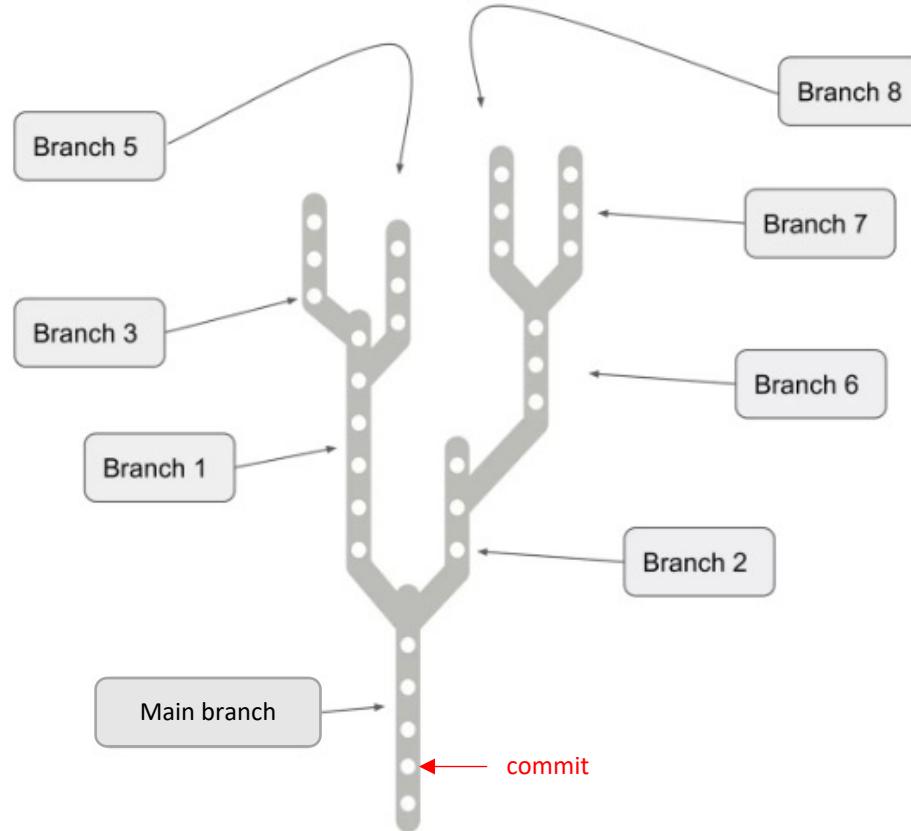
Branches

- What is a branch?
- Creating branches (git branch)
- Merging branches (git merge)
- Resolving merge conflicts

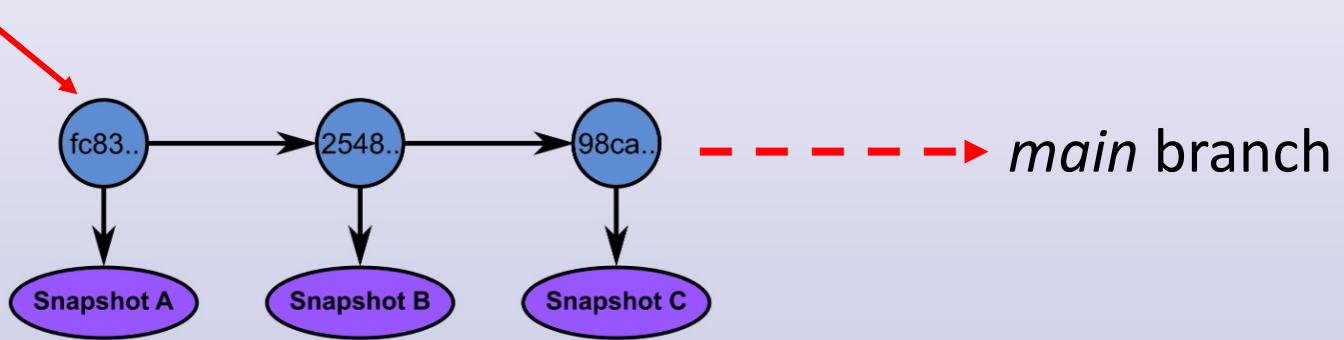


Branches

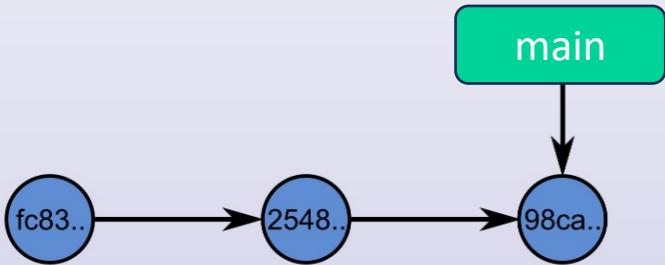
- What is a branch?
- Creating branches (`git branch`)
- Merging branches (`git merge`)
- Resolving merge conflicts



Commit with unique SHA1

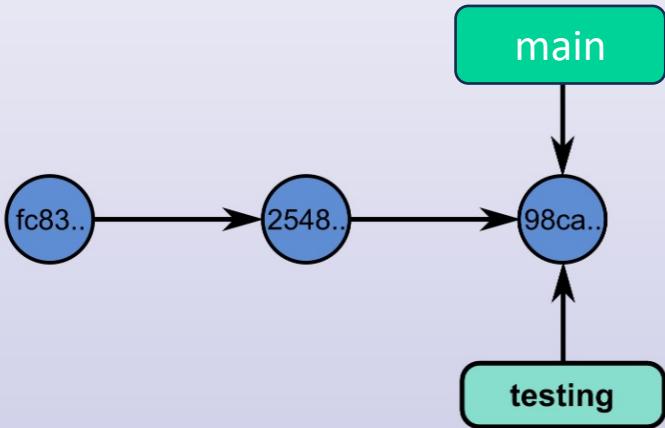


Commits are repository snapshots



main is a branch
created during *git init*

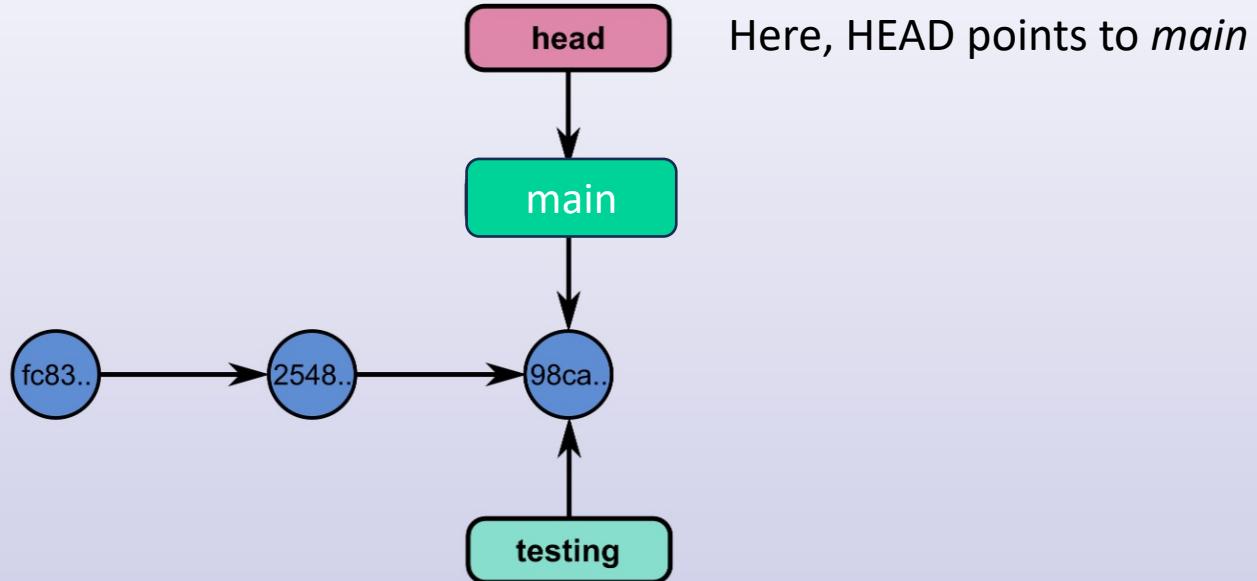
A branch is a pointer to a commit



Create branch

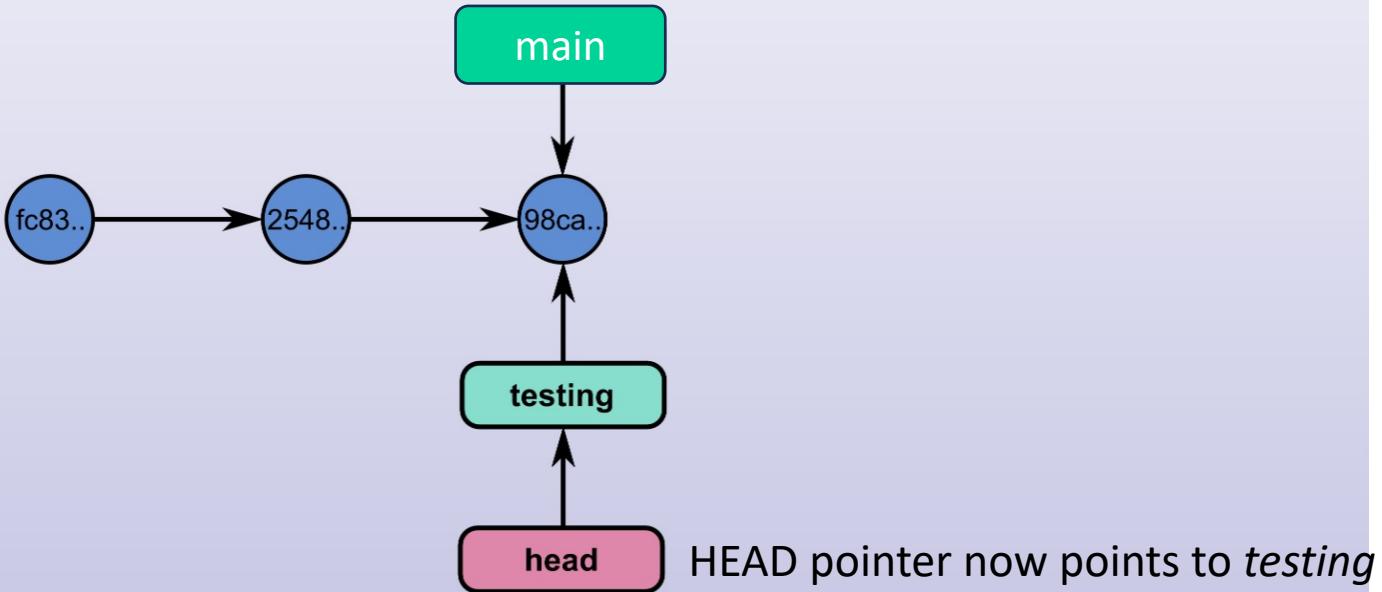
git branch testing

We can have many branches.



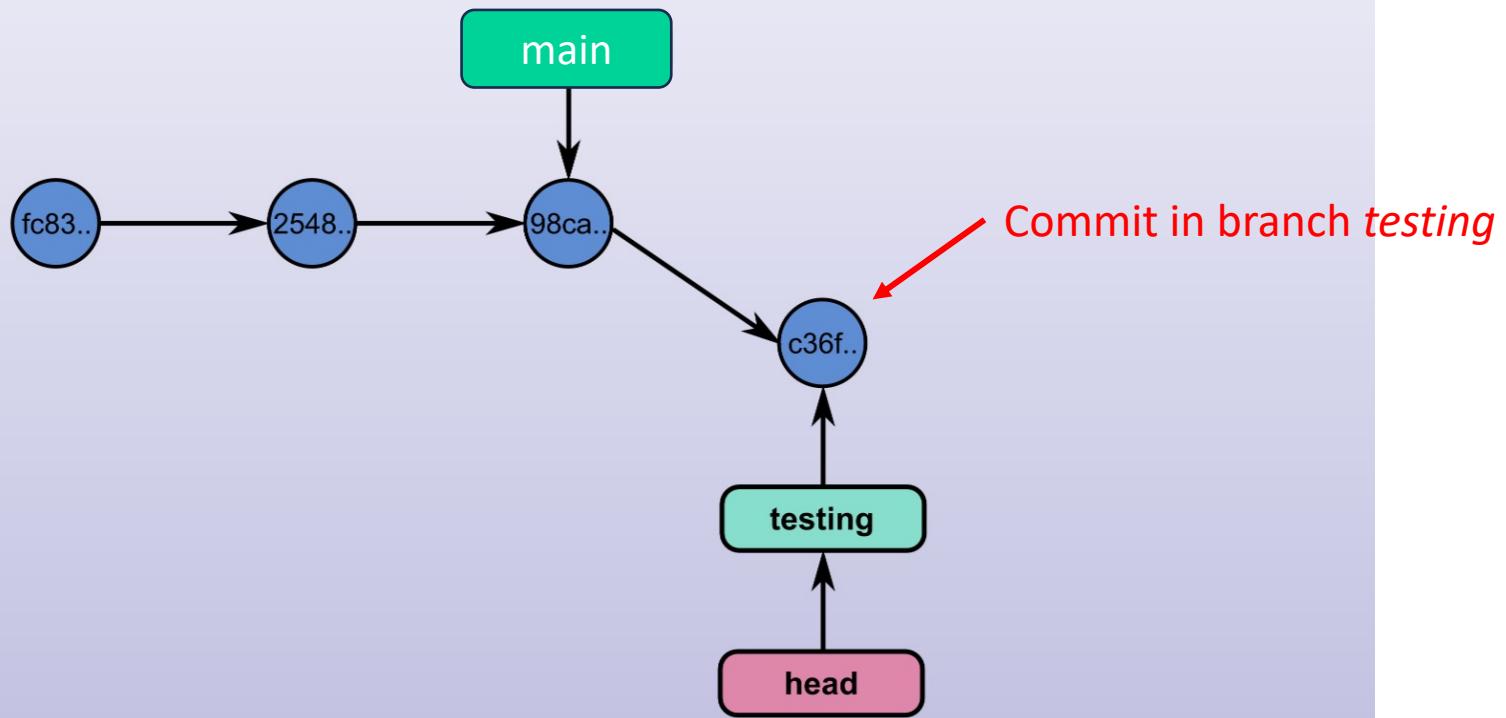
Here, HEAD points to *main*

How do we know what branch we are on? HEAD pointer

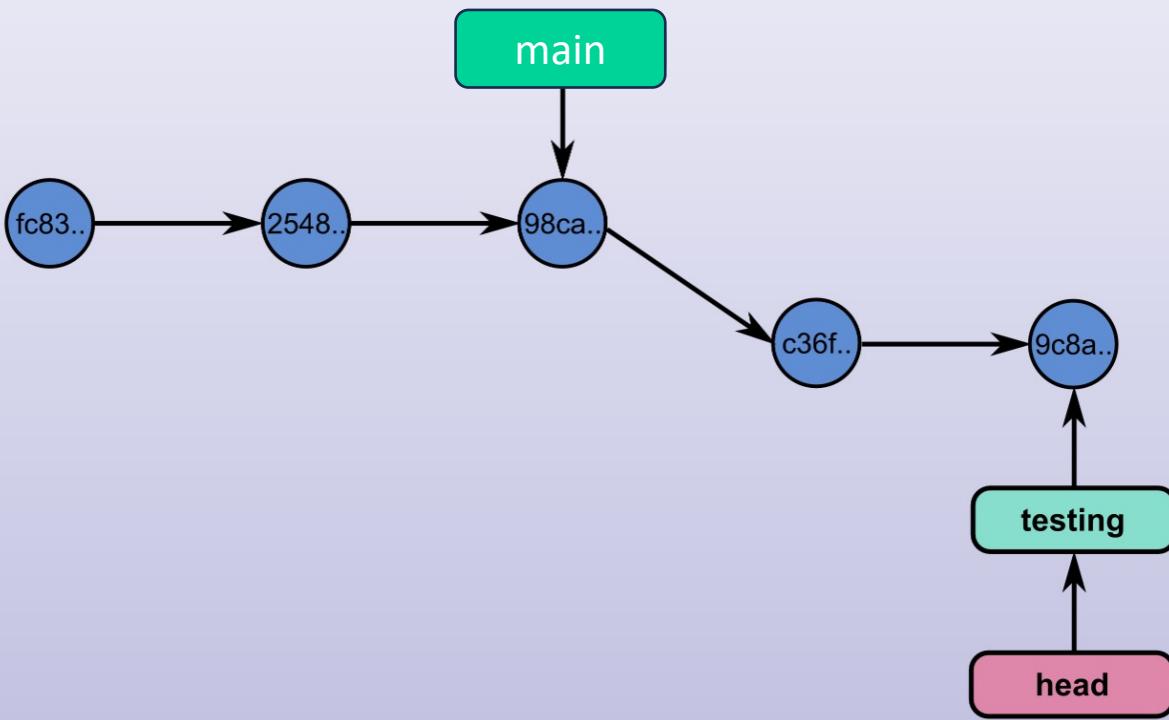


To switch branches
git checkout testing
or
git switch testing

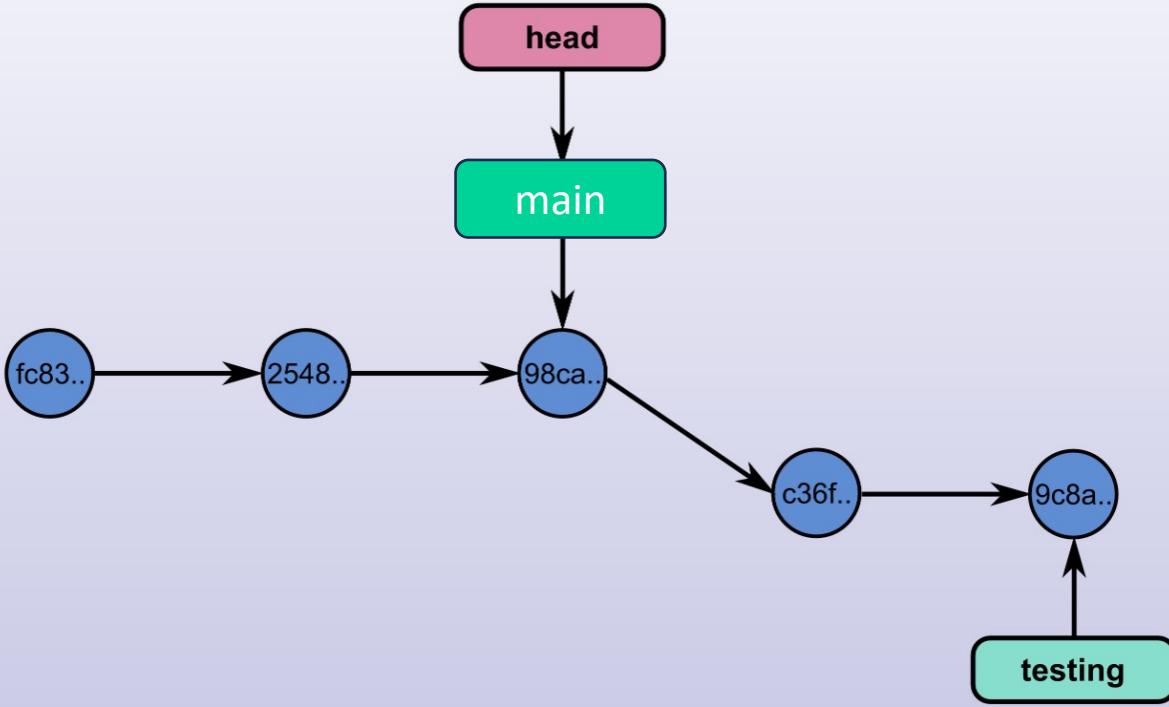
We can switch branches



We can commit in a branch



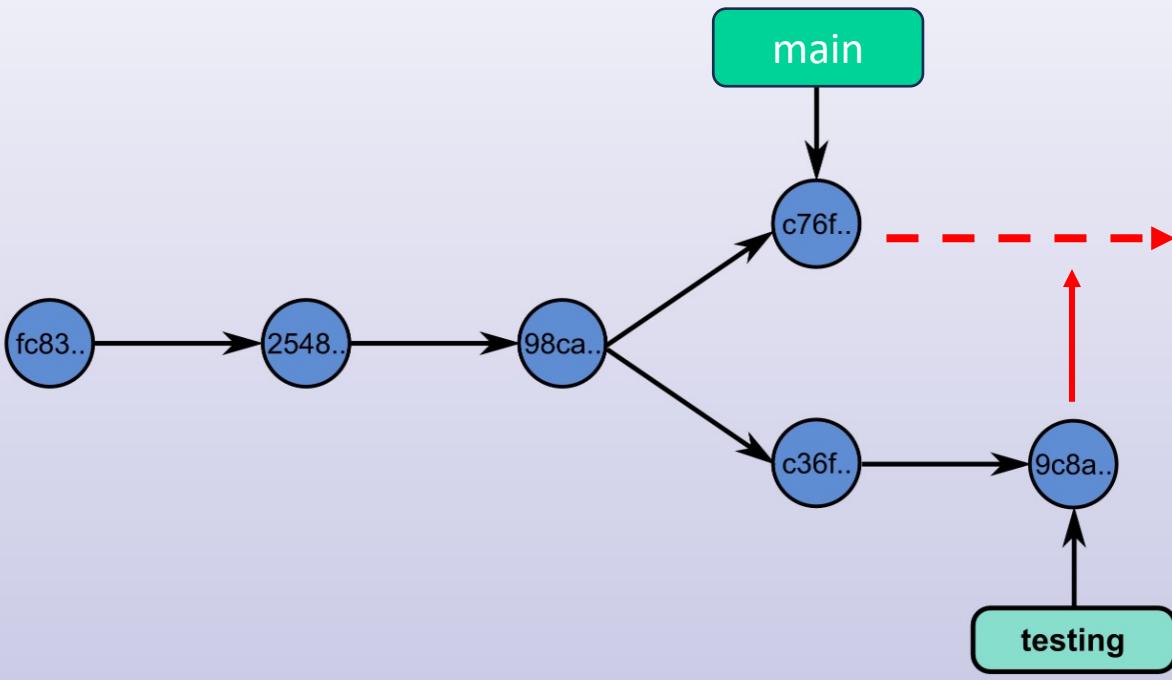
And commit again...



Back to main

git checkout main

Then we can switch branches



Merge feature branch

git merge testing

Branches can diverge...we can merge *testing* into *main*

Branch Commands

\$ git branch

Lists all local branches in the current repository

\$ git branch [branch-name]

Creates a new branch

\$ git branch -d [branch-name]

Deletes the specified branch

\$ git checkout [branch-name]

Switches to the specified branch and updates the working directory

\$ git checkout -b [branch-name]

Create and switch to the specified

\$ git merge [branch]

Combines the specified branch's history into the current branch

Git branches

Exercise

Run these commands

```
git branch  
git branch -a
```

Create a branch named *testing* as follows

```
git branch testing  
git branch -a
```

Switch to that branch and check what branch you are on

```
git checkout testing  
git branch
```

Now create a Python script called *ints.py* with the following code:



```
for k in range(11):  
    print(k)
```

Add and commit. Switch back to *main* branch and check what branch you are on and run

```
git branch  
ls ← What do you not see?
```

Git branches

Exercise

Try deleting the branch *develop* as follows

git branch -d develop

What happens?

Merge the branch *testing* into *main* as follows:

git merge testing

Now run

ls

git branch -d testing
git branch -a

Merge Conflicts

```
echo "add some content" > example.txt  
git add example.txt  
git commit -m "Initial commit"
```

```
git checkout -b branch_to_merge ← Create a feature branch
```

```
echo "different content to merge later" > example.txt  
git add example.txt  
git commit -m "Edit the content of example.txt to make a conflict"
```

```
git checkout main  
git merge branch_to_merge ← Conflicts!
```



This could be work being done by a different developer

Resolve Conflicts

Run

git status

To resolve a "merge conflict" you need to edit the conflicted file.

vim example.txt

Decide what changes to keep. Then:

git add example.txt
git commit -m "Resolve conflict in example.txt"

After you are done merging you should remove the feature branch

git branch -d branch_to_merge

Git Commands

Create repositories

\$ git init [project-name]

Creates a new local repository with the specified name

\$ git clone [url]

Downloads a project and its entire version history

Make changes

\$ git status

Lists all new or modified files to be committed

\$ git add [file]

Snapshots the file in preparation for versioning

\$ git reset [file]

Unstages the file, but preserve its contents

\$ git diff

Shows file differences not yet staged

\$ git diff --staged

Shows file differences between staging and the last file version

\$ git commit -m "[descriptive message]"

Records file snapshots permanently in version history

Group changes

\$ git branch

Lists all local branches in the current repository

\$ git branch [branch-name]

Creates a new branch

\$ git checkout [branch-name]

Switches to the specified branch and updates the working directory

\$ git merge [branch]

Combines the specified branch's history into the current branch

\$ git branch -d [branch-name]

Deletes the specified branch

Review history

\$ git log

Lists version history for the current branch

\$ git log --follow [file]

Lists version history for a file, including renames

\$ git diff [first-branch]...[second-branch]

Shows content differences between two branches

\$ git show [commit]

Outputs metadata and content changes of the specified commit