

Introduction to Version Control with Git

Fall 2024 ASTG Training

Motivation

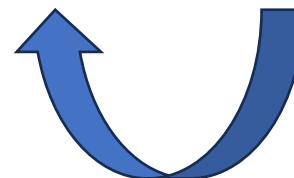


```
wrf.F
Users > ccruz > scratch > nu-wrf-dev > WRF > main > wrf.F
  ...
  ! the top-level domain, extract from initial or restart data, setting up time-keeping, and
  ! then calling the <a href=integrate.html>integrate</a> routine to advance the domain
  ! to the ending time of the simulation. After the integration is completed, the model
  ! is properly shut down.
  !
!</DESCRIPTION>
IMPLICIT NONE
#ifndef _OPENMP
CALL setfeenv()
#endif
! Set up WRF model.
CALL wrf_init
!
! Run digital filter initialization if requested.
CALL wrf_dfi
!
#if ( WRFPLUS == 1 )
! Run adjoint check and tangent linear check if requested.
CALL wrf_adtl_check
#endif
!
! WRF model time-stepping. Calls integrate().
#if ( WRFPLUS == 1 )
IF ( config_flags%dyn_opt .E0. dyn_em ) &
#endif
CALL wrf_run
!
#if ( WRFPLUS == 1 )
! WRF model time-stepping. Calls integrate().
IF ( config_flags%dyn_opt .E0. dyn_em_tl .and. config_flags%tl_standalone ) &
CALL wrf_run_tl_standalone
!
! WRF model time-stepping. Calls integrate().
IF ( config_flags%dyn_opt .E0. dyn_em_ad ) &
#endif
CALL wrf_run_ad
```

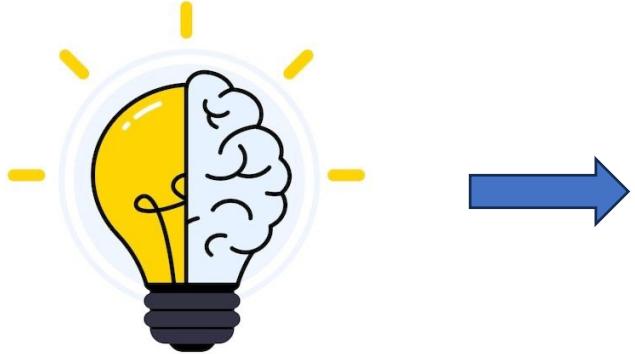
Ln 61, Col 1 Spaces: 2 UTF-8 LF Plain Text



Motivation

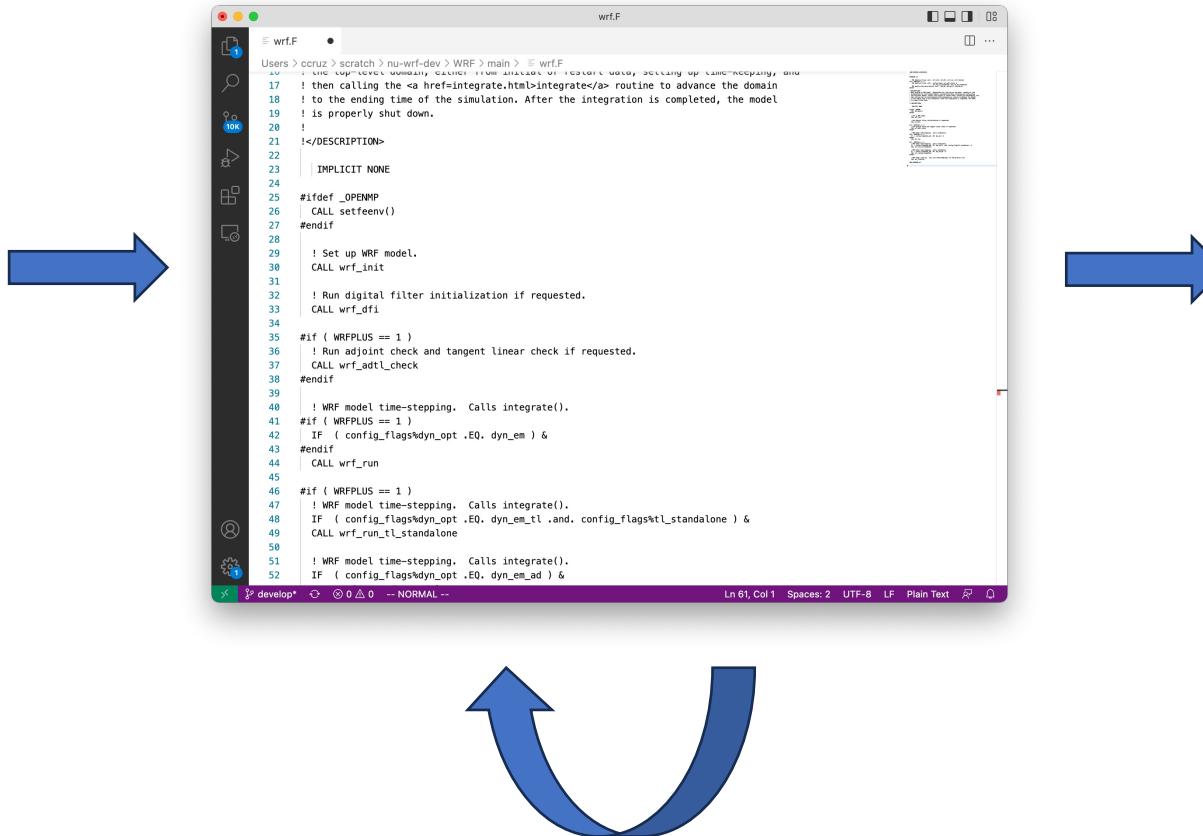
A screenshot of a terminal window titled "wrf.F". The window displays a large amount of Fortran code for the WRF model. The code includes comments explaining the purpose of various sections, such as setting up the domain, calling the "integrate" routine, and handling adjoint and tangent linear checks. The code is organized into several sections with labels like "</DESCRIPTION>" and "#IMPLICIT NONE". The terminal interface shows standard Unix-style navigation keys at the bottom.

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 various conditional statements (#if, #endif), subroutine calls (CALL wrf_init, CALL wrf_adtl_check), and comments explaining the model's behavior. The background of the terminal has a faint watermark of the Earth.

Software development includes the continuous process of modifying programs

Proper code management is essential for effective and sustainable software development



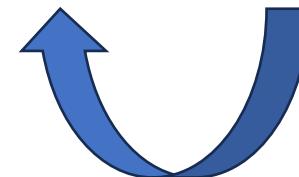
Software development includes the continuous process of modifying programs

Proper code management is essential for effective and sustainable software development

Requires a version
control system (**VCS¹**)

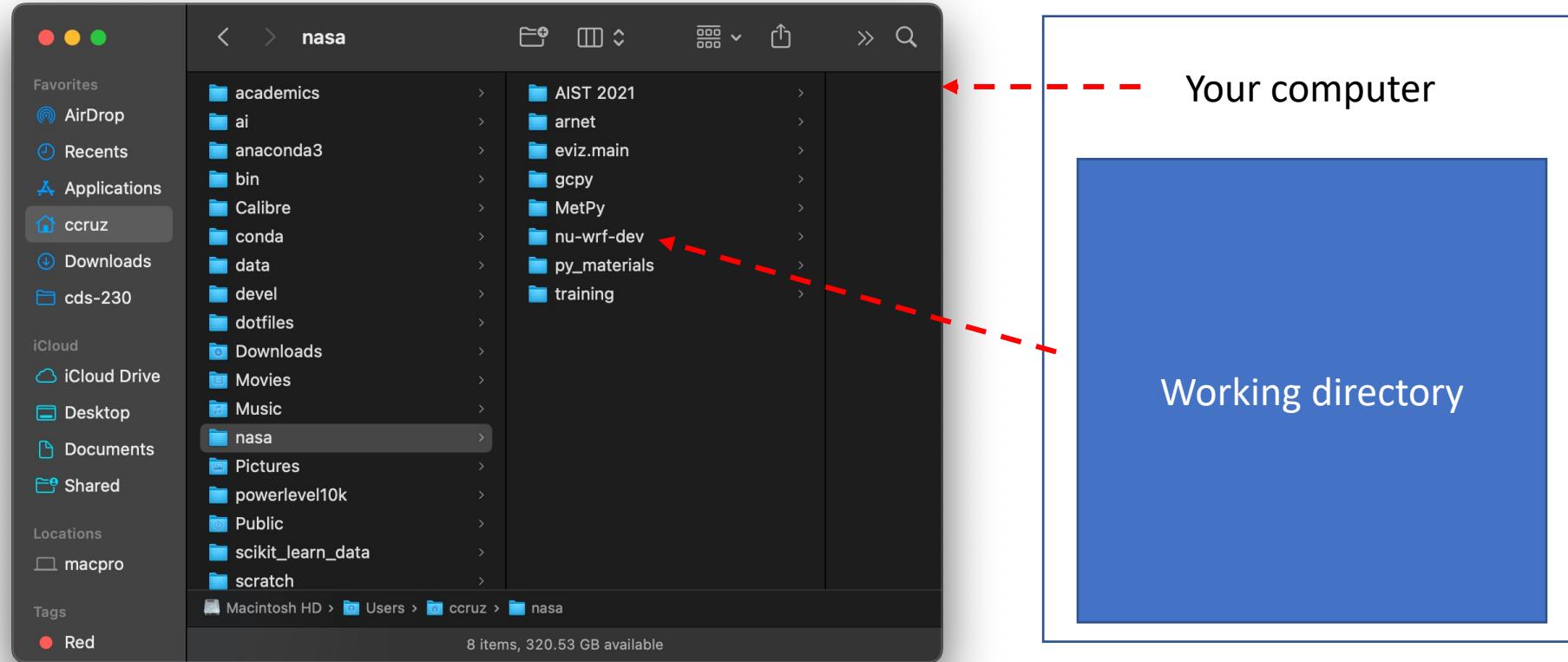


The screenshot shows a terminal window titled 'wrf.F' displaying a portion of a Fortran source code file. The code is related to the WRF (Weather Research and Forecasting) model. It includes comments explaining the purpose of various sections, such as setting up the domain and calling the 'integrate' routine. The code uses standard Fortran syntax with IF statements and CALL directives. The terminal interface includes a sidebar with icons for file operations and a status bar at the bottom.



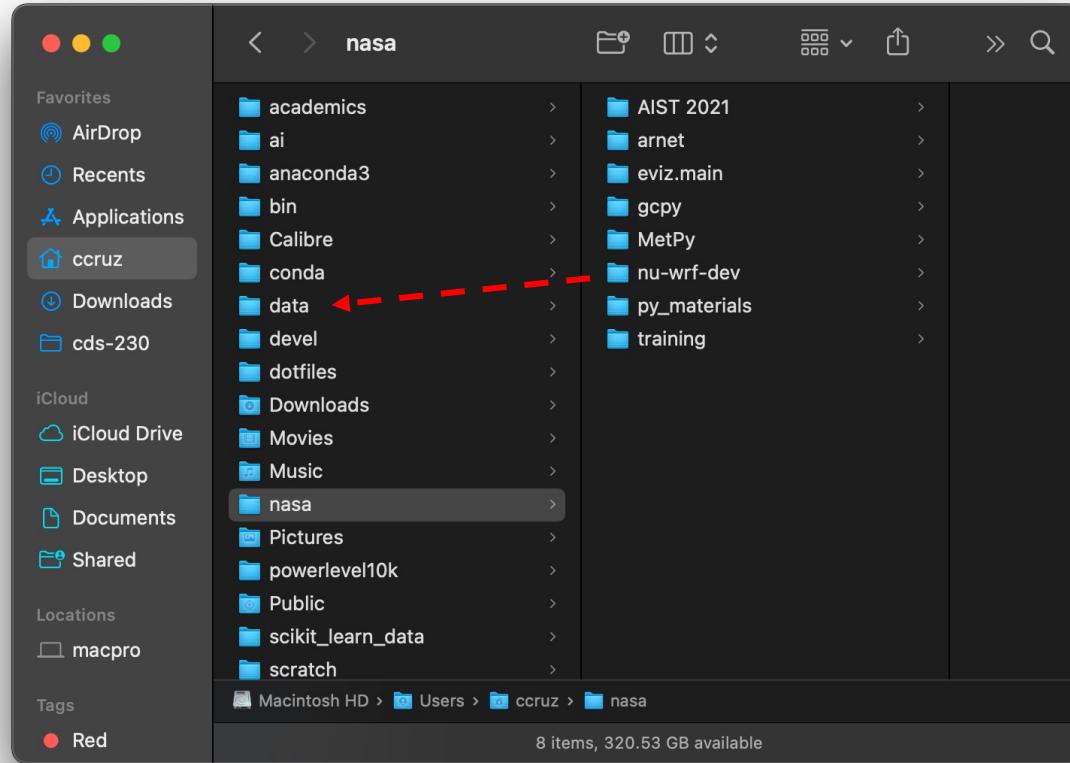
¹Another technical term is Source Code Management (SCM)

How do we manage source code changes *without* a VCS?



How do we manage source code changes *without* a VCS?

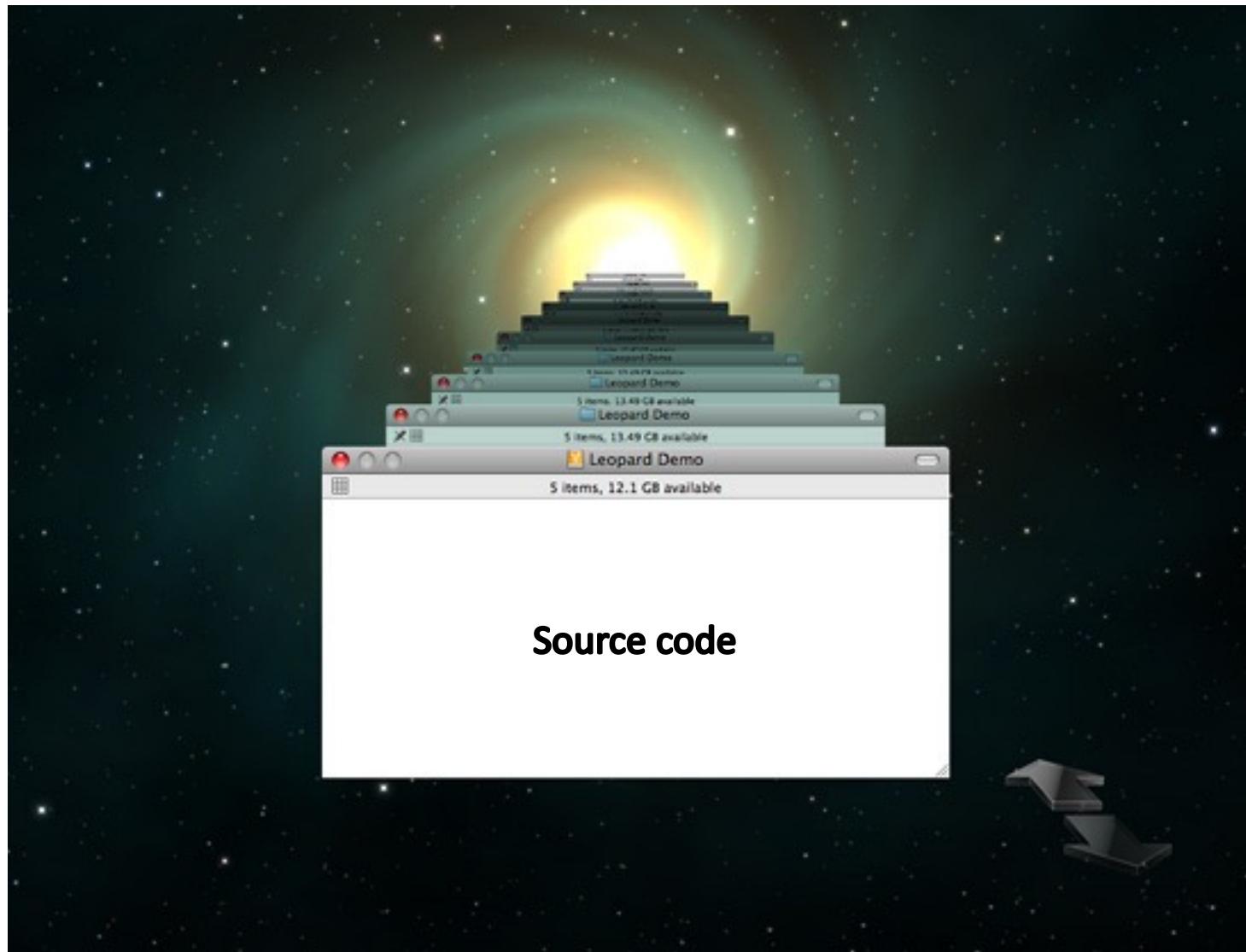
A personal “version control system”



Your computer

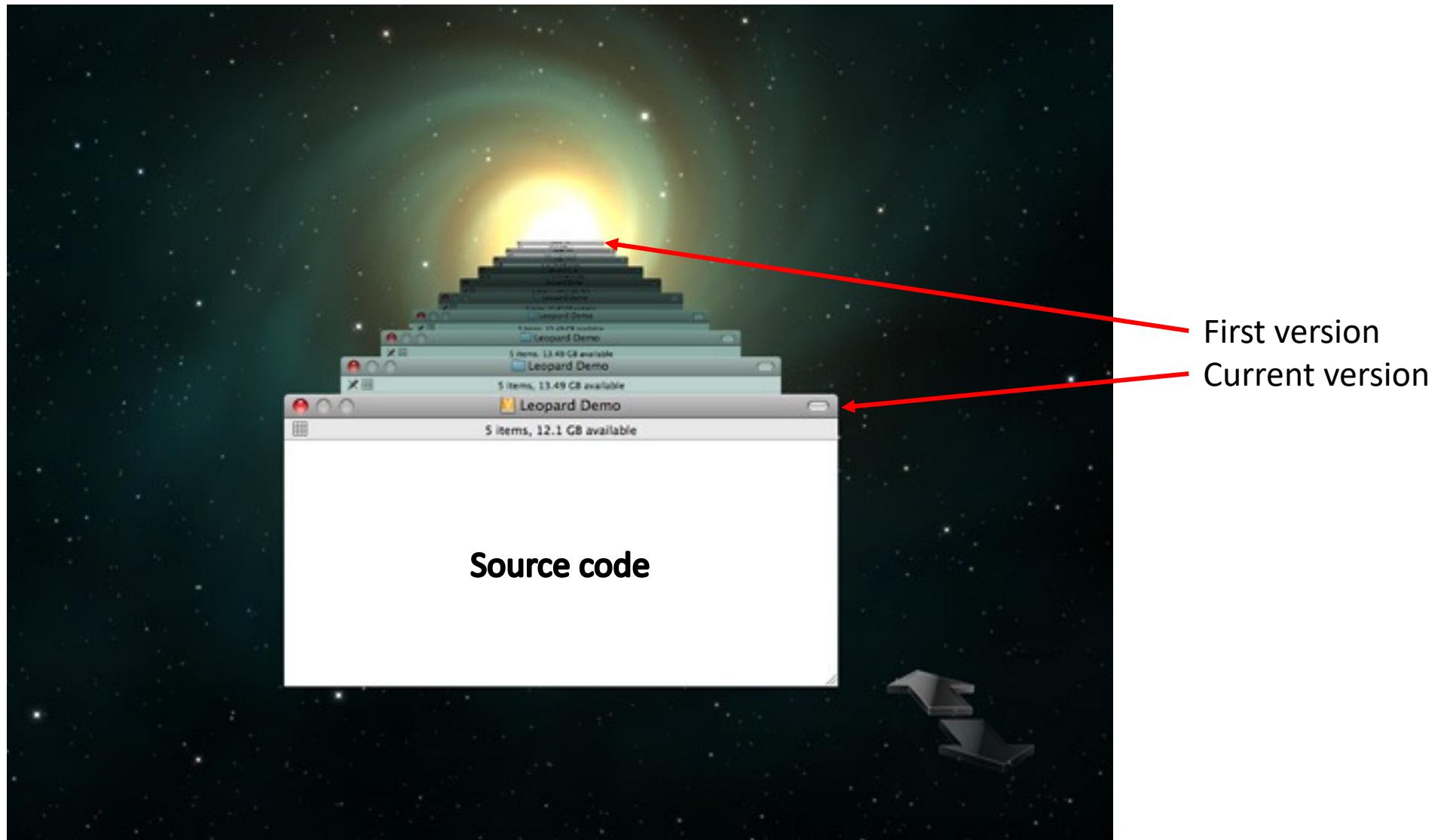
```
[ccruz: data] $  
script.py.v4  
script.py.v3  
script.py.v2  
script.py.v1  
version_notes
```

Version Control System (VCS)

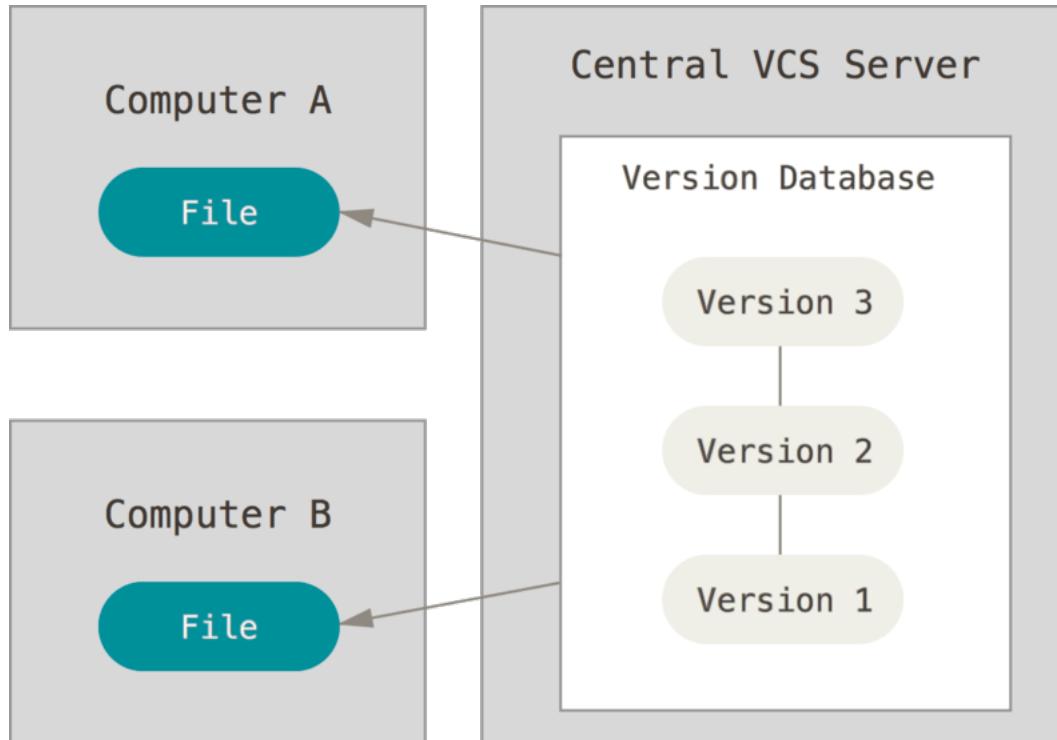


Source code

Version Control System (VCS)



Centralized VCS



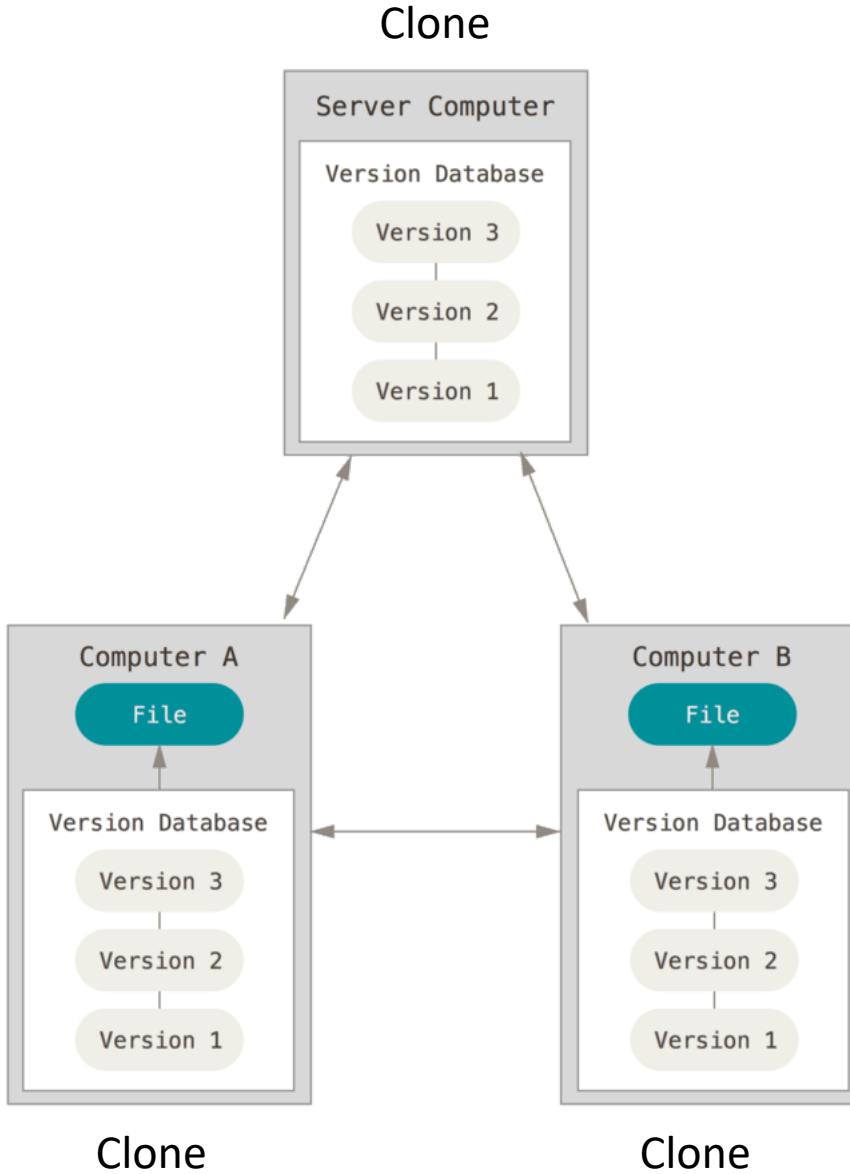
Examples:

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 at <https://git-scm.com/>. The top navigation bar includes the Git logo and the tagline "fast-version-control". A search bar is located in the top right corner. The main content area features two sections: "About" and "Documentation". The "About" section highlights Git's advantages over other systems like Subversion, CVS, Perforce, and ClearCase. The "Documentation" section links to command reference pages, the Pro Git book, and other materials. Below these are sections for "Downloads" (with links to Mac, Windows, and Linux clients) and "Community" (with links to bug reporting, mailing lists, and chat). A sidebar on the left contains a link to the "Pro Git" book. The right side of the page displays a diagram illustrating Git's distributed nature, showing multiple repositories connected by bidirectional arrows. A large monitor icon in the center shows the latest source release (2.46.0) and a "Download for Mac" button. At the bottom, links are provided for "Mac GUIs", "Tarballs", "Windows Build", and "Source Code".

git --fast-version-control

Type / to 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.

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

Latest source Release
2.46.0
[Release Notes \(2024-07-29\)](#)

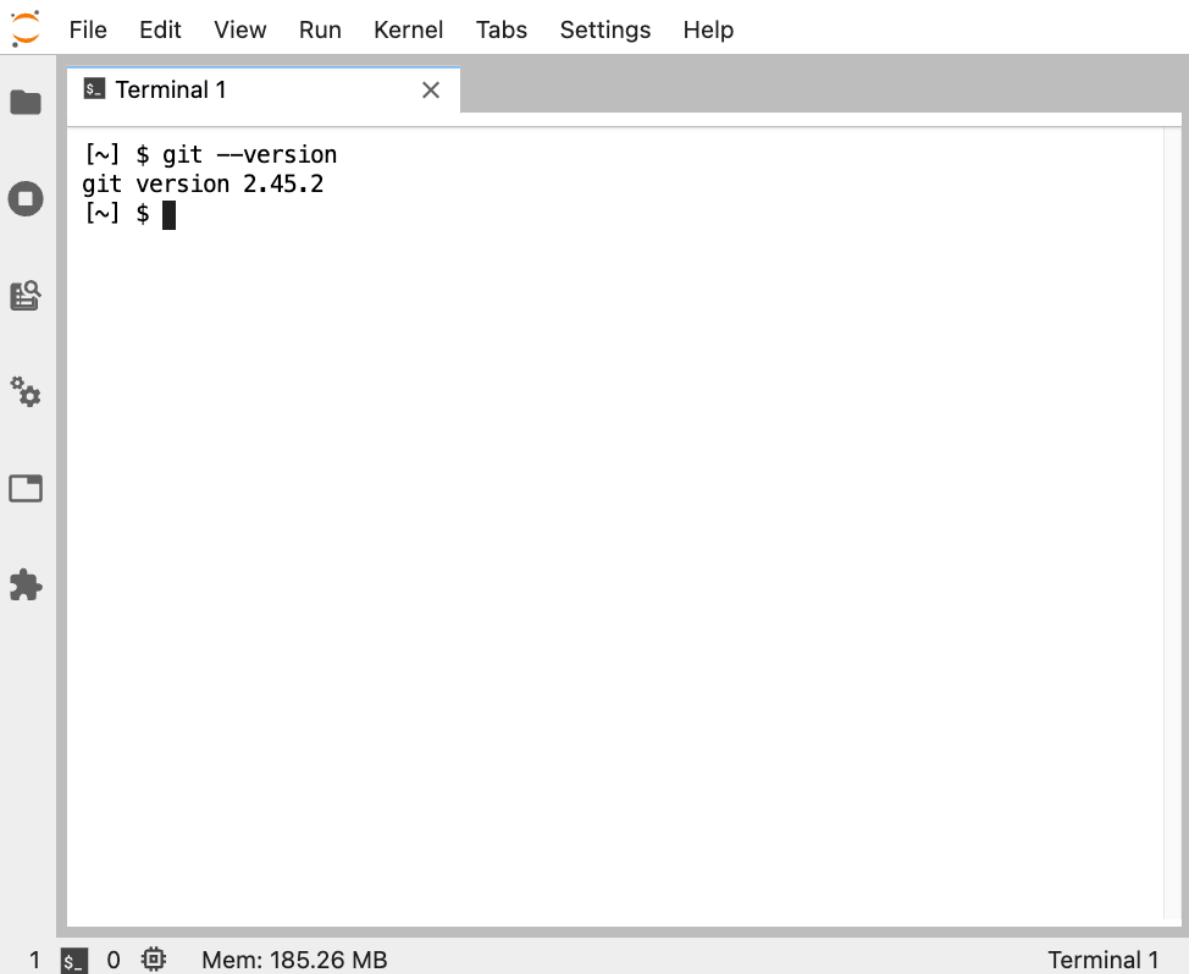
[Download for Mac](#)

[Mac GUIs](#) [Tarballs](#)
[Windows Build](#) [Source Code](#)

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 dark theme with light-colored text. At the top, there is a menu bar with options: File, Edit, View, Run, Kernel, Tabs, Settings, and Help. Below the menu is a toolbar with several icons: a folder, a magnifying glass, a gear, a square, and a star. The main area of the terminal shows the command "git --version" being run and its output: "git version 2.45.2". The bottom of the window displays system status information: "1 \$ 0 Mem: 185.26 MB" and "Terminal 1".

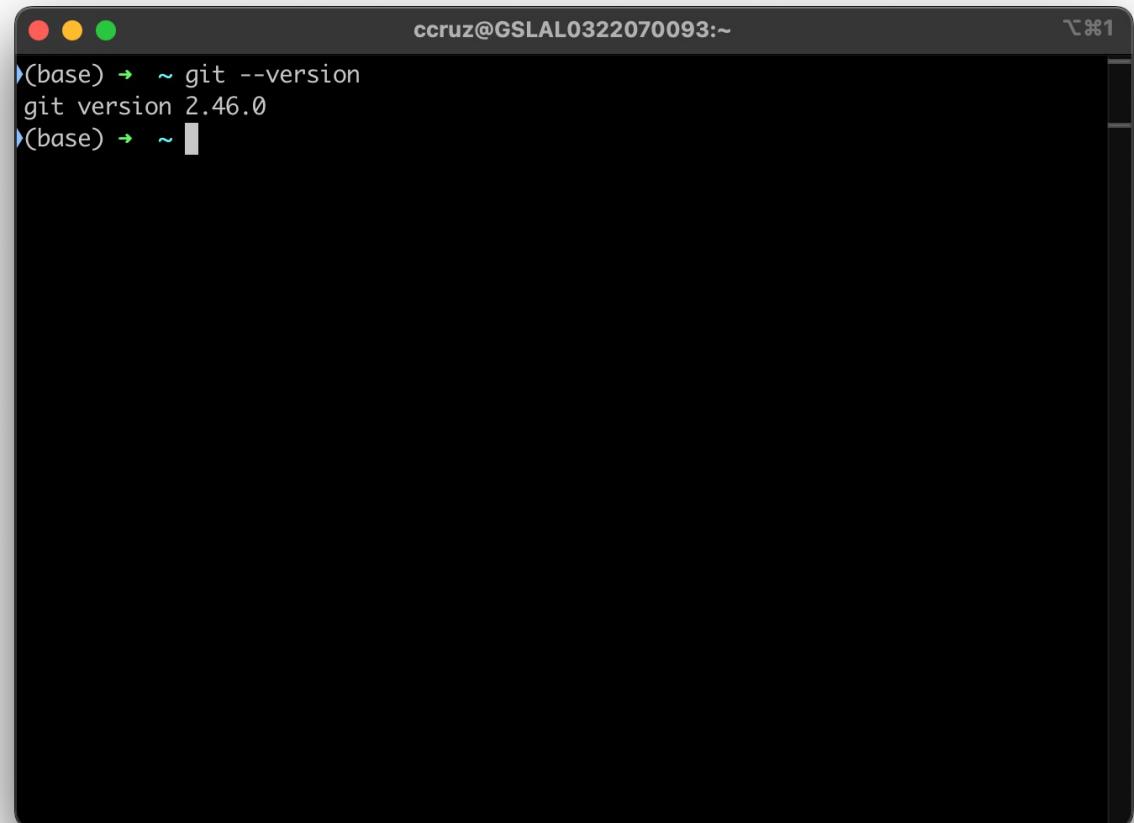
SMCE training platform: <https://training.astg.smce.nasa.gov/hub/login>

Using Git

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



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



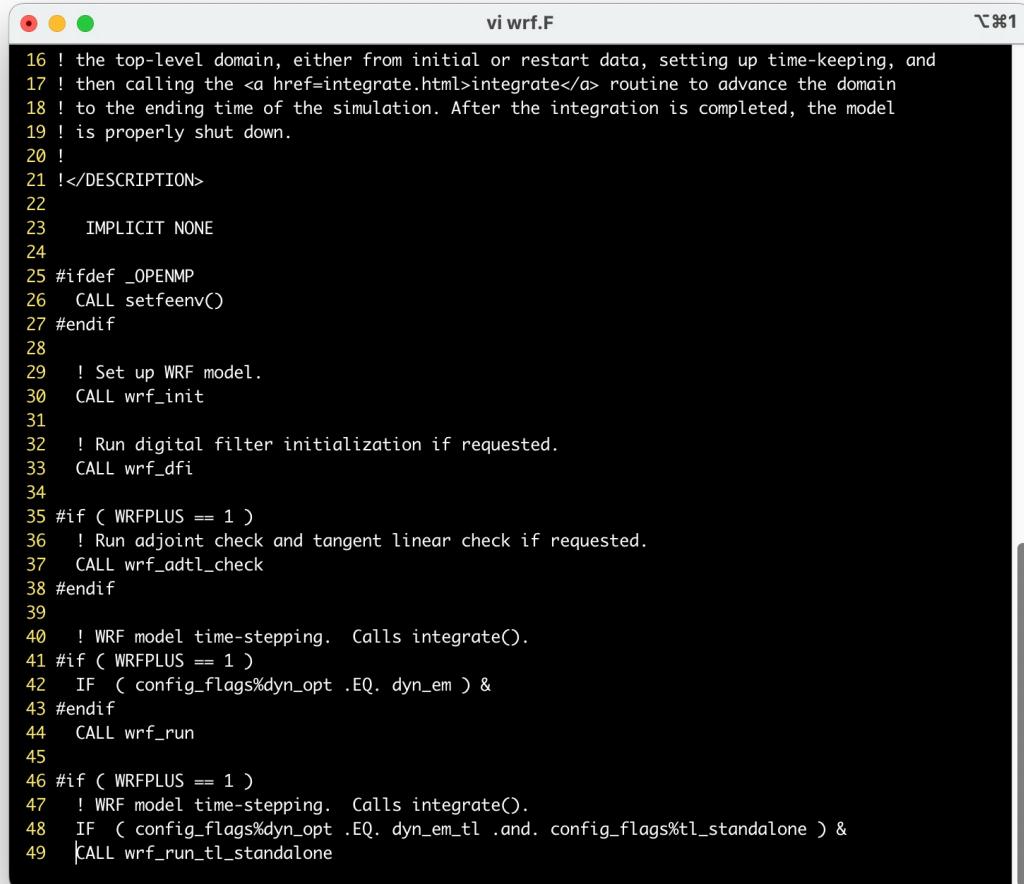
A screenshot of a terminal window on a Mac OS X system. The window title bar shows the user's name and host as "ccruz@GSLAL0322070093:~". The main pane of the terminal displays the command "git --version" followed by its output: "git version 2.46.0". The terminal has a dark theme with red, yellow, and green window controls.

```
(base) ~ git --version
git version 2.46.0
(base) ~
```

Creating and modifying source code

Code Editors: vim, emacs, nano, etc.

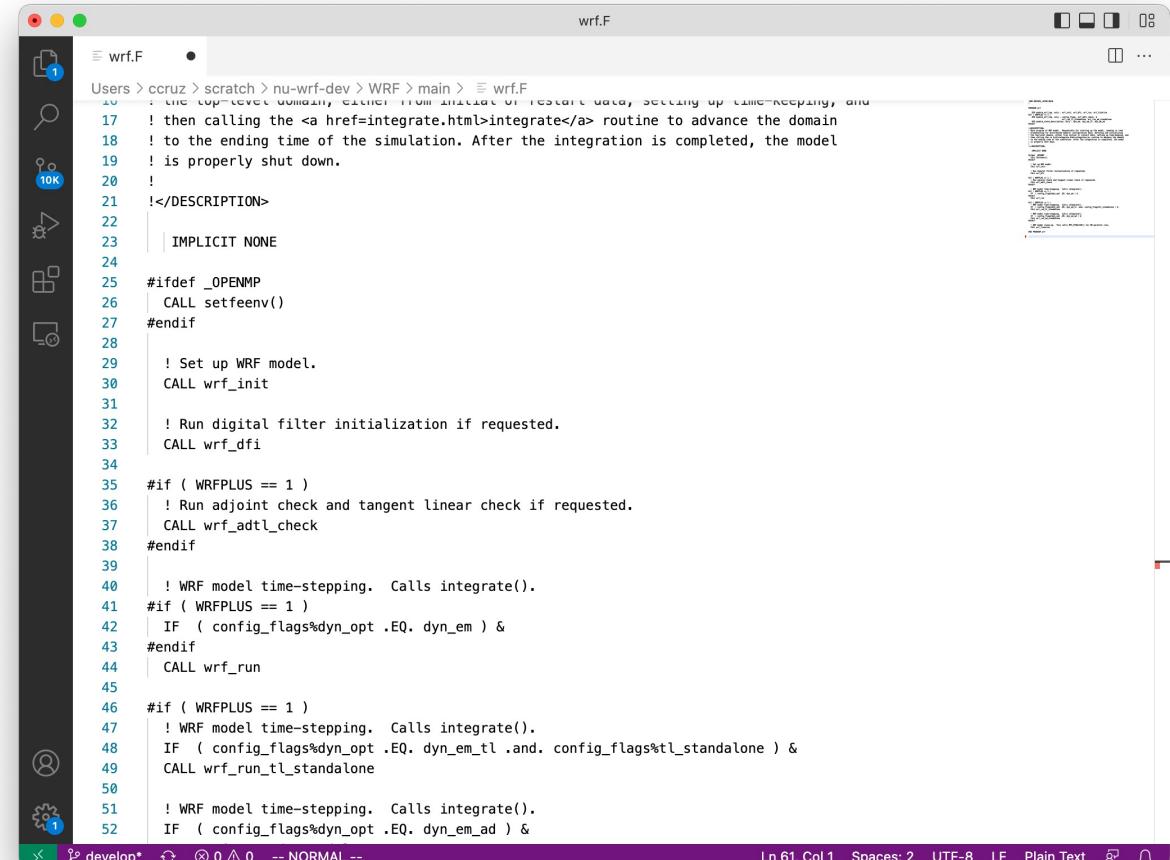
IDEs: VS Code, PyCharm, Sublime*, etc.



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 with a dark background and light text. The title bar says "ccruz@GSLAL0322070093:~". The command "git --version" is run, and the output shows "git version 2.46.0". Below the command, there is a prompt "(base) ~". In the center of the window, the text "git config --help" is displayed in a larger, italicized font.

```
ccruz@GSLAL0322070093:~  
❯(base) ~ git --version  
git version 2.46.0  
❯(base) ~
```

git config --help

- **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

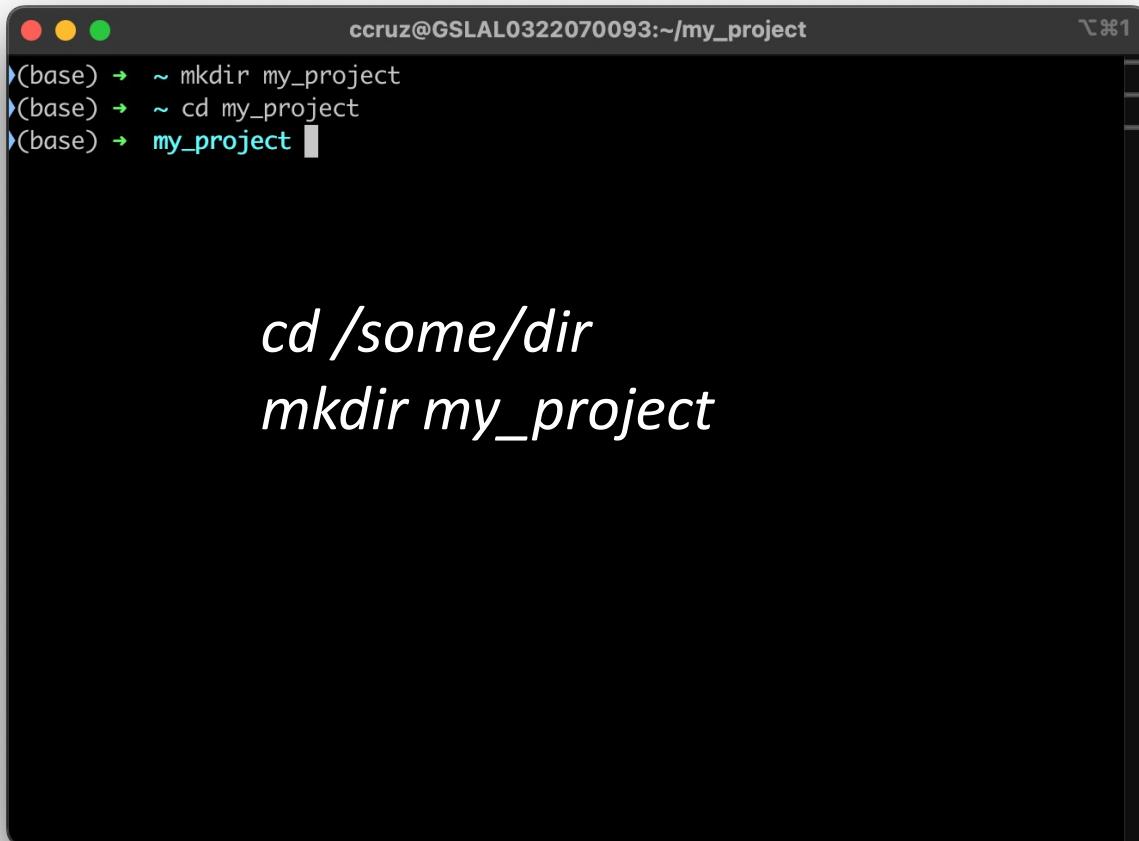
[core]

editor = /usr/bin/vim

Create a Working Directory*

Exercise

run the following commands on a terminal:

A screenshot of a macOS terminal window. The title bar says "ccruz@GSLAL0322070093:~/my_project". The window contains three lines of text: "(base) ~ mkdir my_project", "(base) ~ cd my_project", and "(base) ~ my_project". The text "(base)" and the prompt "~" are in light gray, while "mkdir", "cd", and "my_project" are in white.

```
(base) ~ mkdir my_project
(base) ~ cd my_project
(base) ~ my_project
```

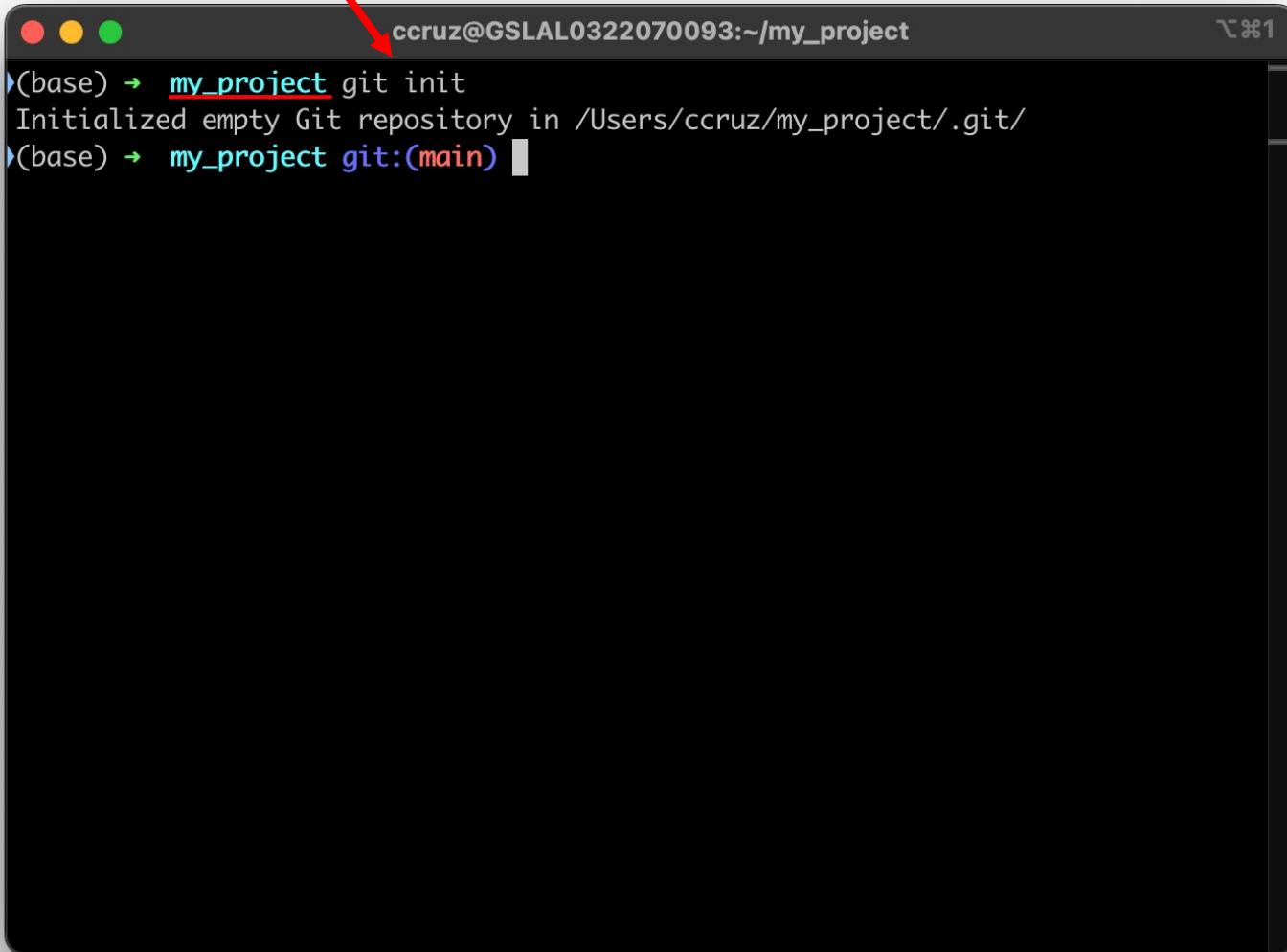
*cd /some/dir
mkdir my_project*

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



A screenshot of a macOS terminal window titled "ccruz@GSLAL0322070093:~/my_project". The window shows the command "git init" being run in a directory named "my_project". A red arrow points from the text "\$ git init" above the terminal to the word "my_project" in the terminal's command line.

```
ccruz@GSLAL0322070093:~/my_project
(base) → my_project git init
Initialized empty Git repository in /Users/ccruz/my_project/.git/
(base) → my_project git:(main)
```

Creating a “repo”

\$ git clone

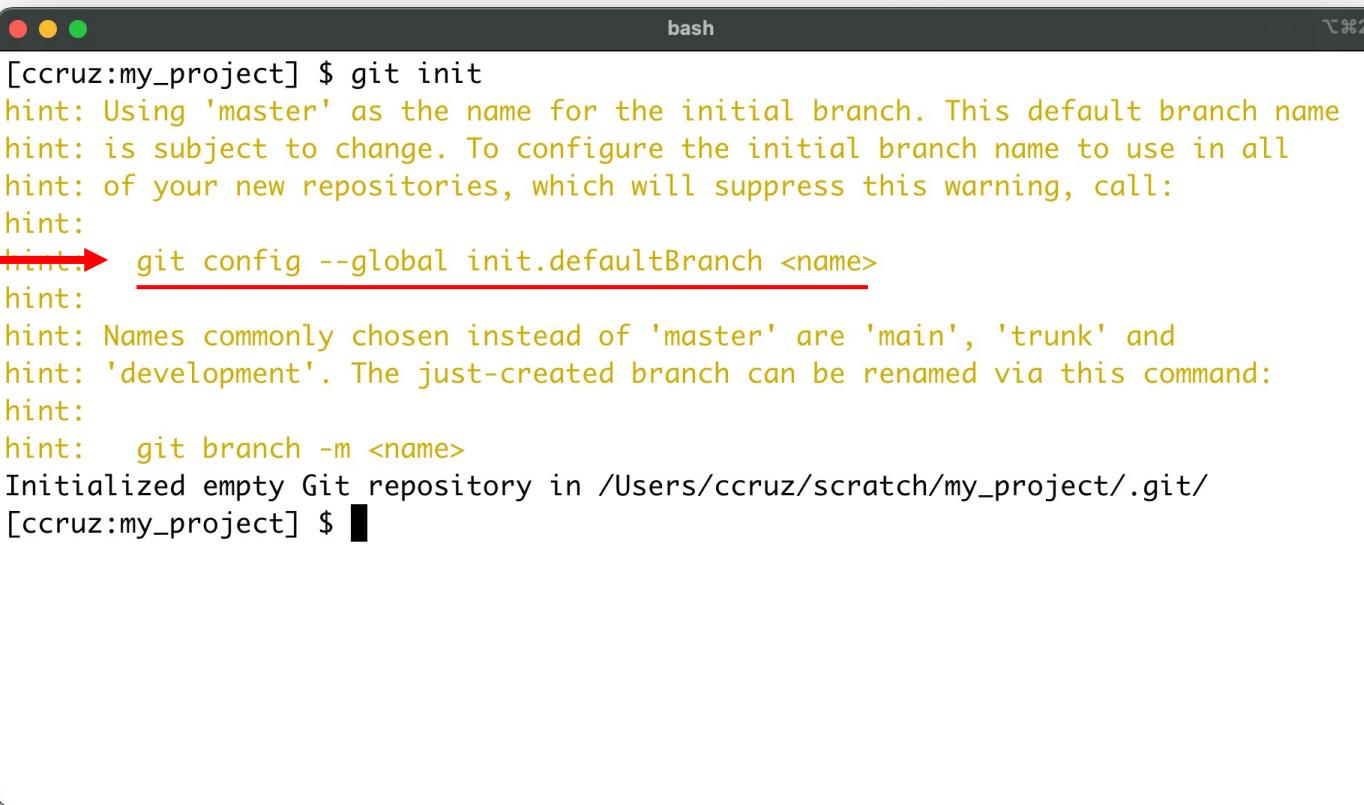


```
ccruz@GSLAL0322070093:~
```

```
(base) ~ git clone https://github.com/NASA-LIS/LISF
Cloning into 'LISF'...
remote: Enumerating objects: 51428, done.
remote: Counting objects: 100% (1863/1863), done.
remote: Compressing objects: 100% (751/751), done.
remote: Total 51428 (delta 1216), reused 1599 (delta 1071), pack-reused 49565 (from 1)
Receiving objects: 100% (51428/51428), 238.70 MiB / 2.74 MiB/s, done.
Resolving deltas: 100% (38448/38448), done.
Updating files: 100% (7199/7199), done.
(base) ~
```

Creating a “repo”

\$ *git init*



The screenshot shows a terminal window with a dark theme. The title bar says "bash". The command [ccruz:my_project] \$ git init is entered. The terminal displays a warning message about the default branch name "master" and provides instructions on how to change it. A red arrow points to the line "hint: git config --global init.defaultBranch <name>". The message concludes with the confirmation of a new repository creation.

```
[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] $
```

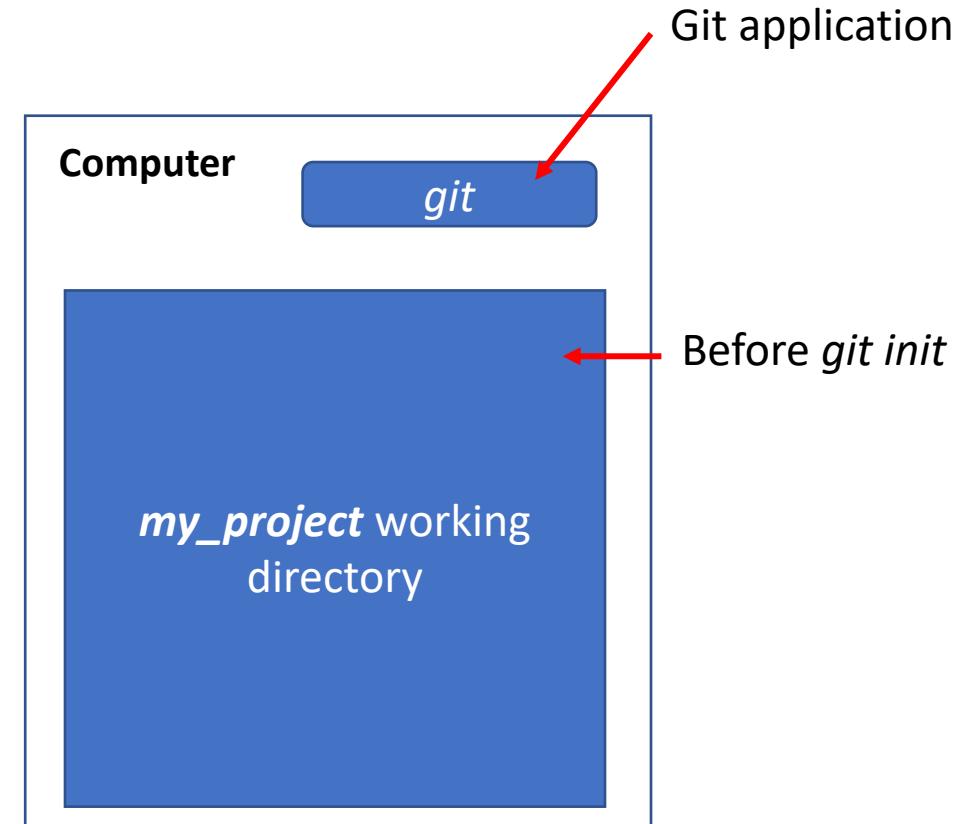
Create an Empty Repository

Exercise

run the following commands on your terminal:

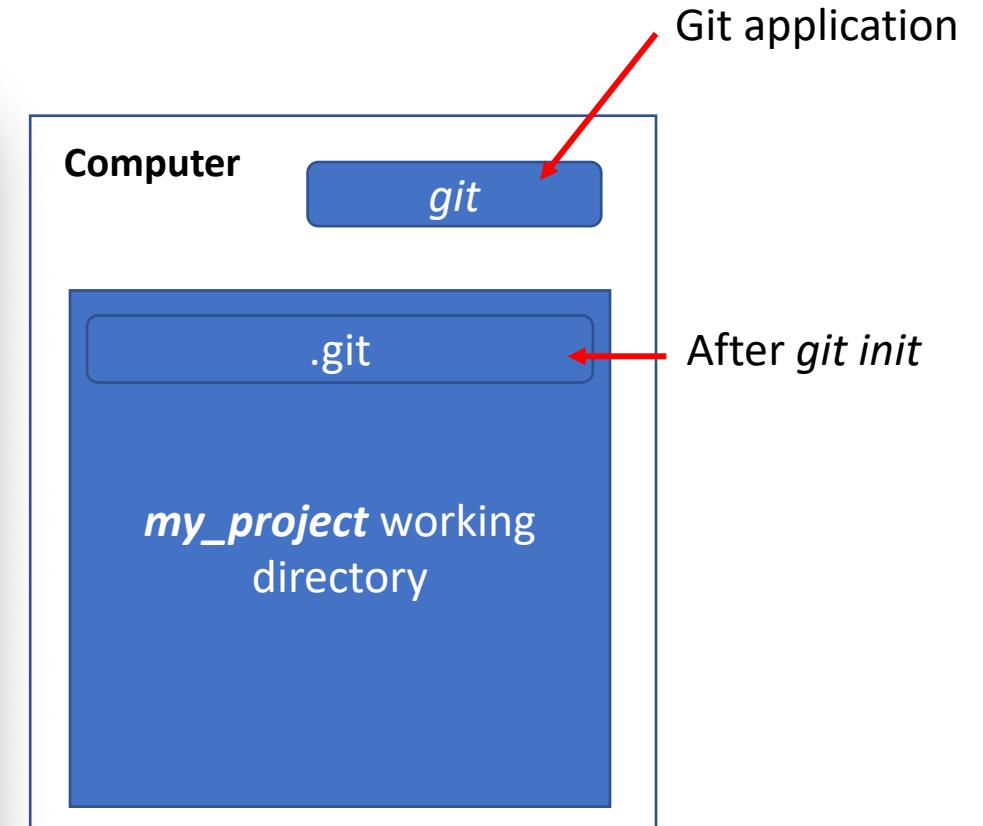


A screenshot of a macOS terminal window. The title bar shows 'ccruz@GSLAL0322070093:~'. The command line shows '(base) ~' with a green arrow icon. Below the prompt, four commands are listed in white text:
*cd my_project
ls -la
git init
ls -la*



Create an Empty Repository

```
ccruz@GSLAL0322070093:~/my_project
(base) ~ cd my_project
(base) my_project git:(main) ls -la
total 0
drwxr-xr-x@ 3 ccruz staff 96 Sep 12 09:54 .
drwxr-x---+ 129 ccruz staff 4128 Sep 12 09:57 ..
drwxr-xr-x@ 9 ccruz staff 288 Sep 12 09:54 .git
(base) my_project git:(main)
```



`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]

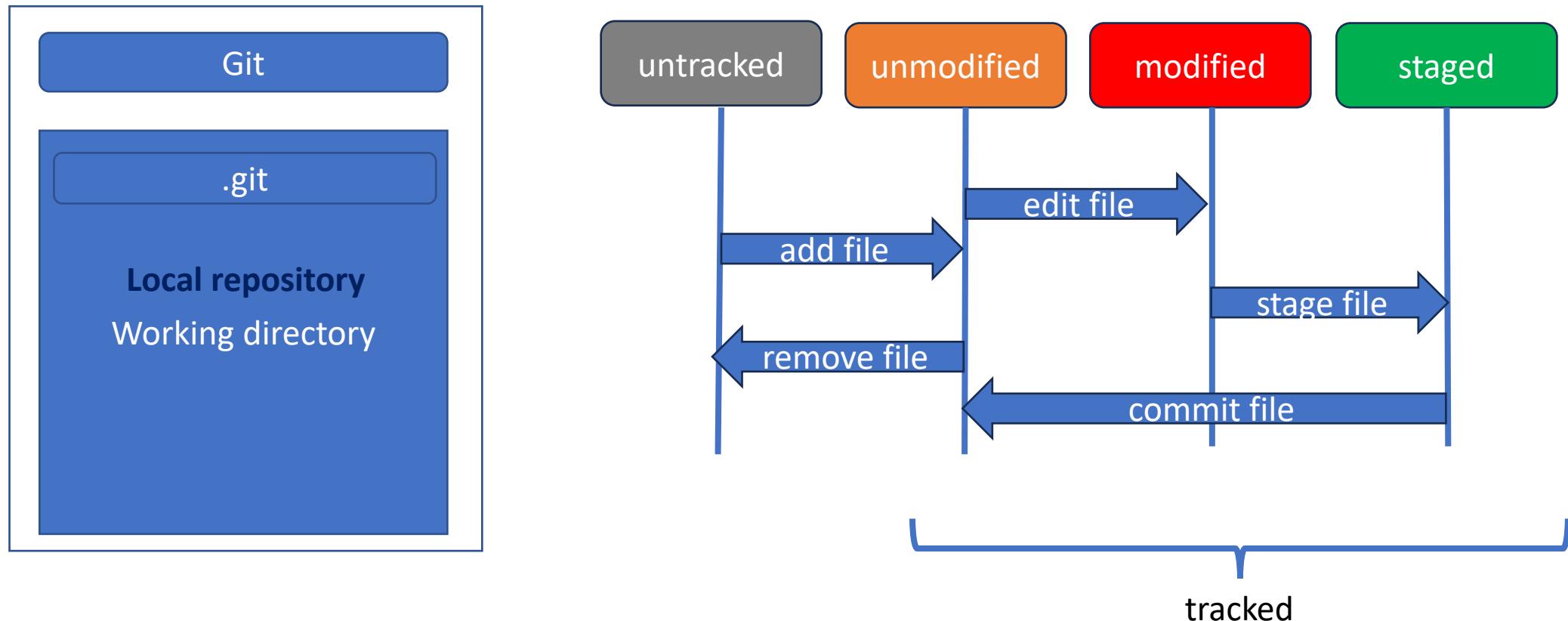
Unstages the file, but preserve its contents

\$ 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 the 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 such as *.DS_Store*
 - IDE-generated directories such as *.idea* and *.vscode*
 - 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
```

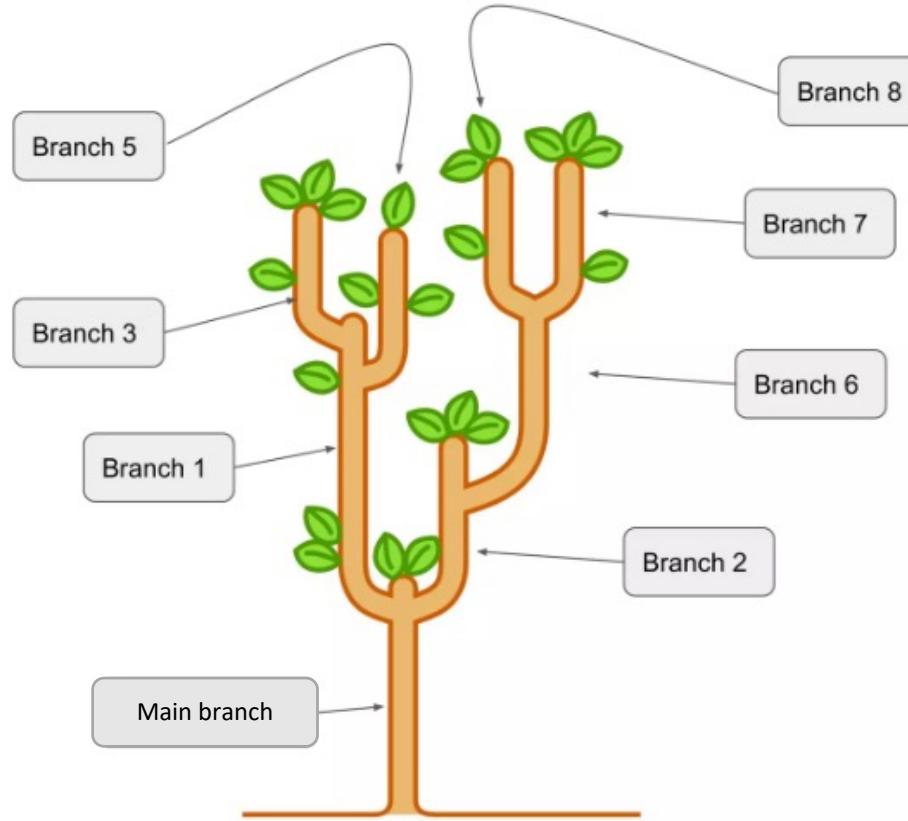
Run the following commands and observe what happens

```
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.

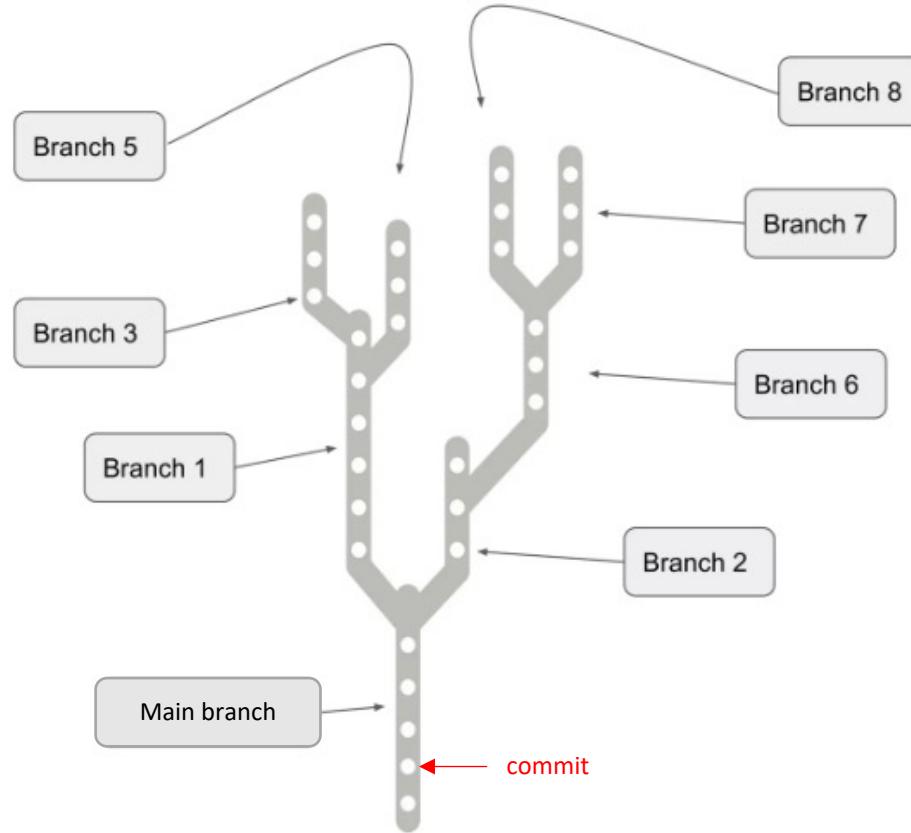
Branches

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

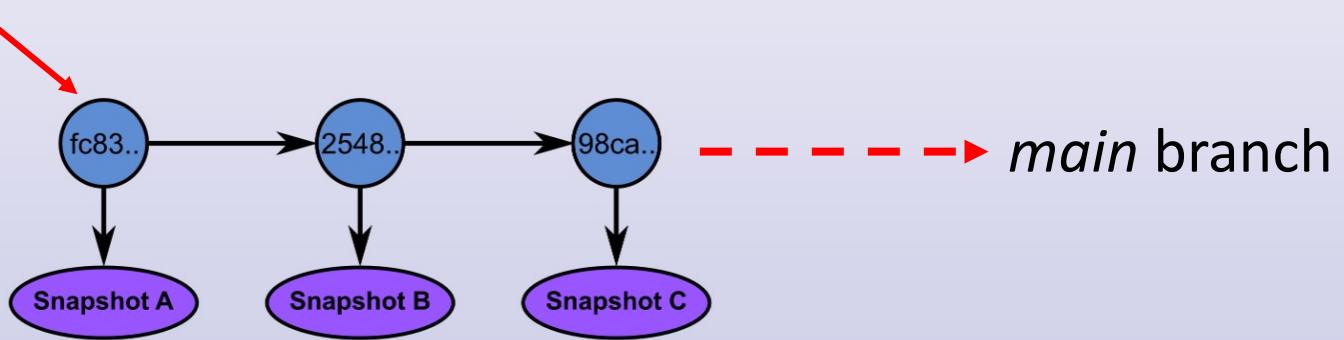


Branches

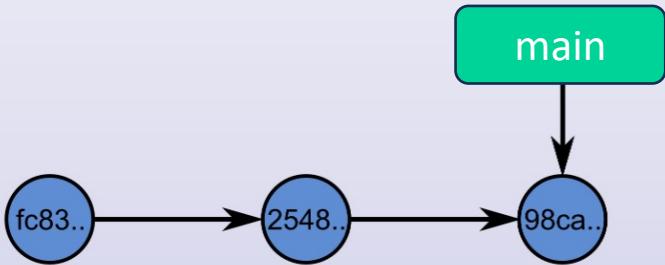
- What is a branch?
- Creating branches (`git branch`)
- Merging branches (`git merge`)
- Resolving merge conflicts



Commit with unique SHA1

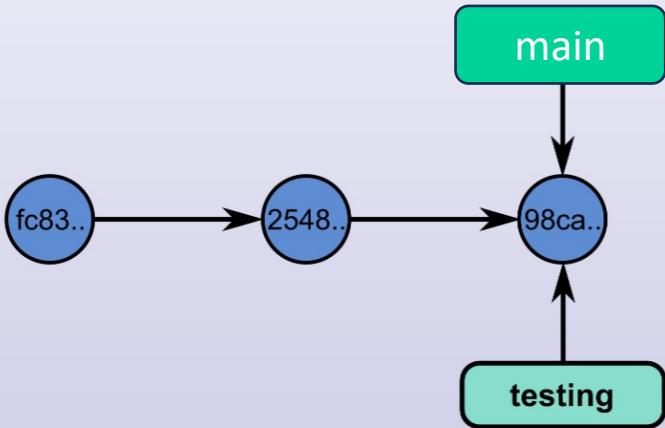


Commits are repository snapshots



main is a branch
created during *git init*

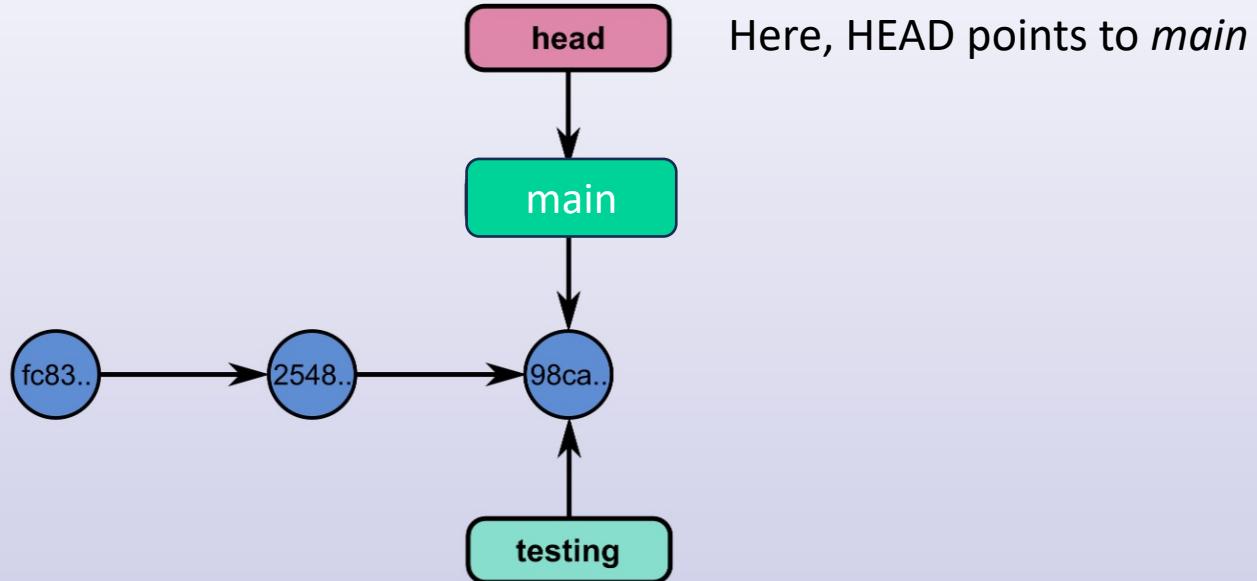
A branch is a pointer to a commit



Create branch

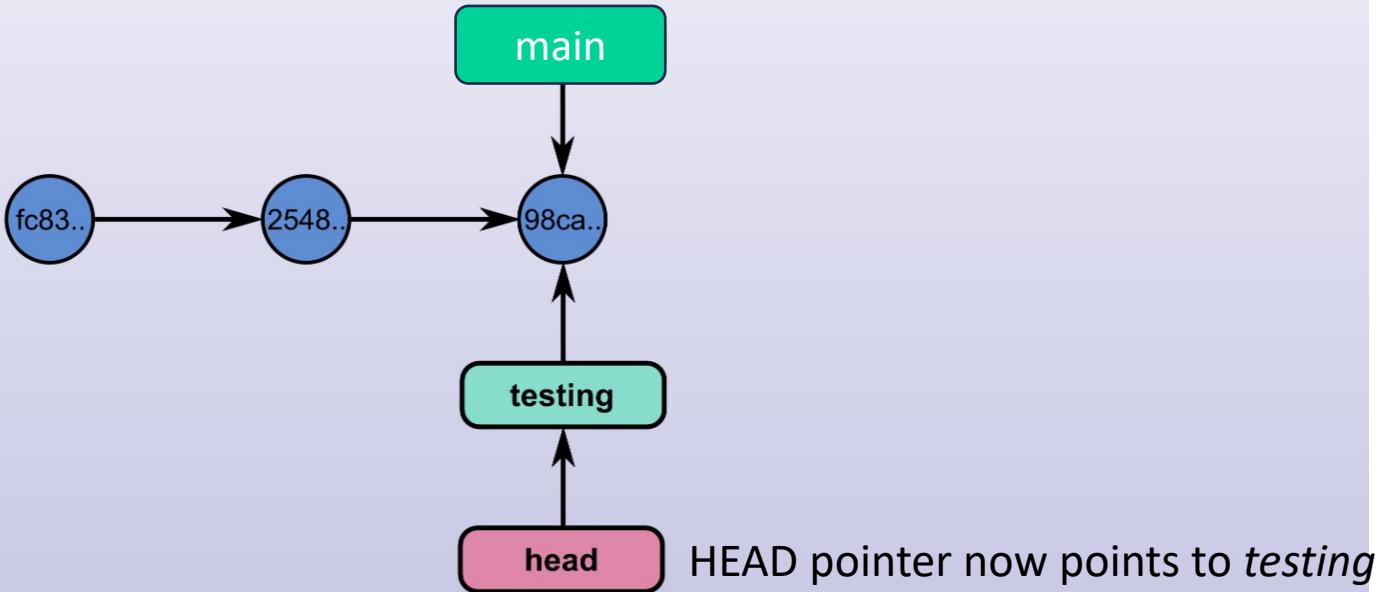
git branch testing

We can have many branches.



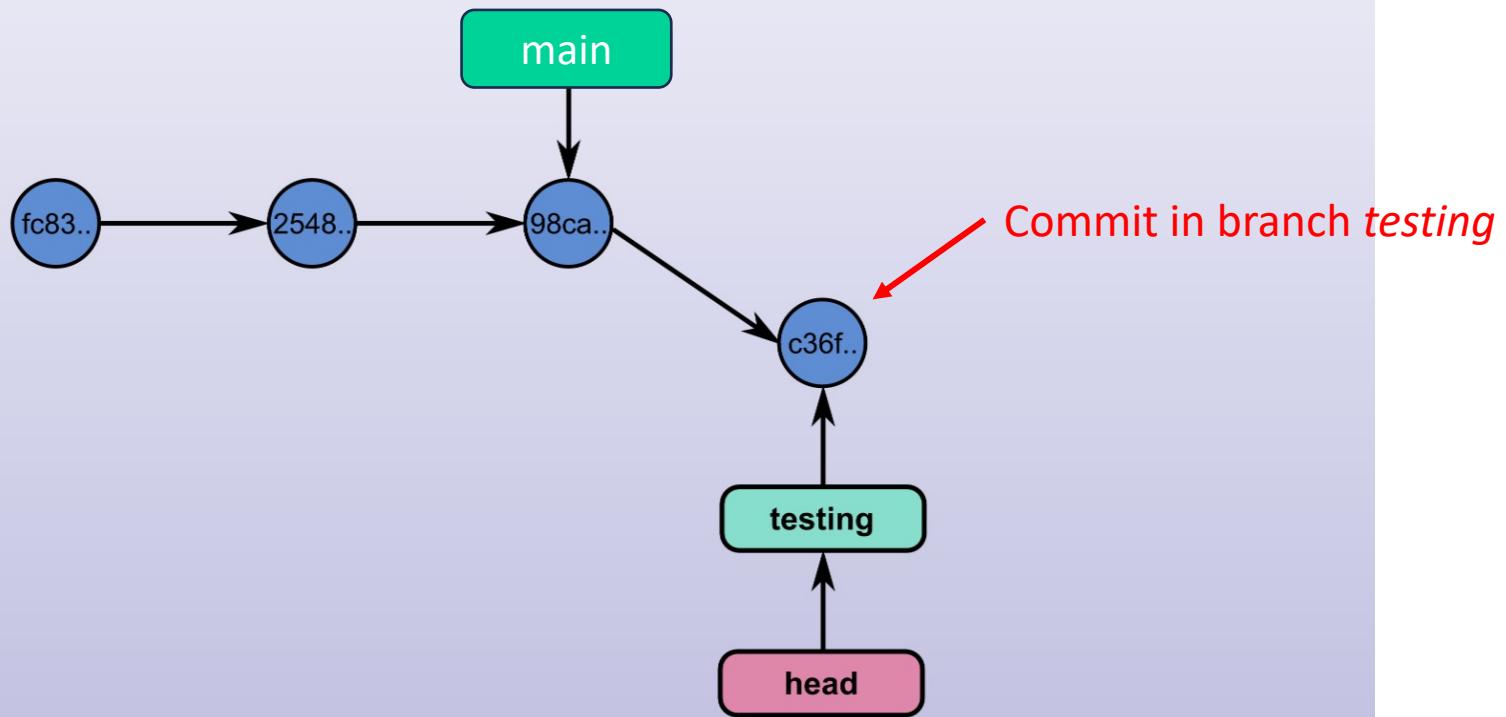
Here, HEAD points to *main*

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

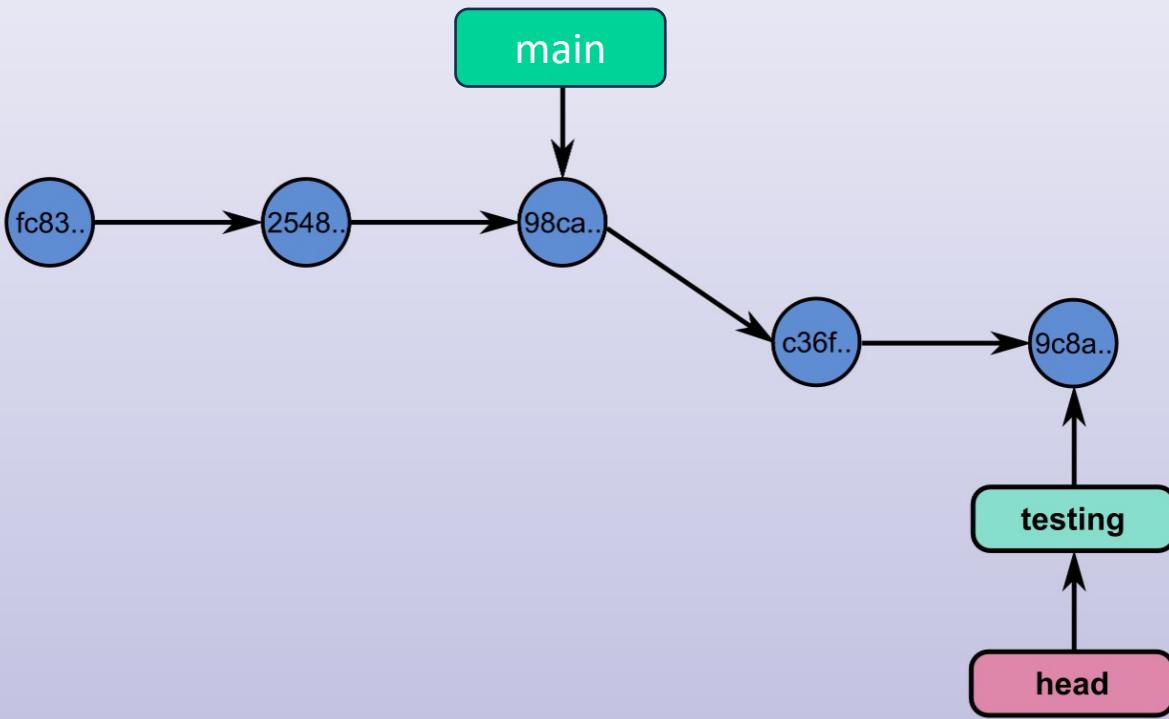


To switch branches
git checkout testing
or
git switch testing

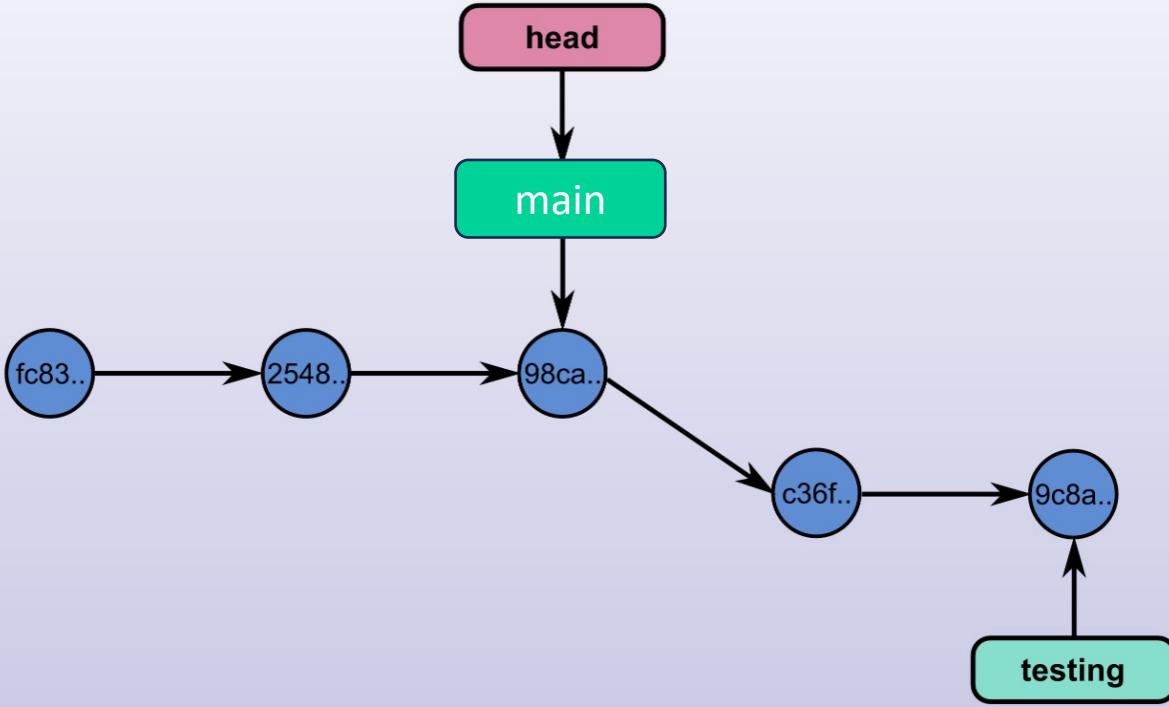
We can switch branches



We can commit in a branch



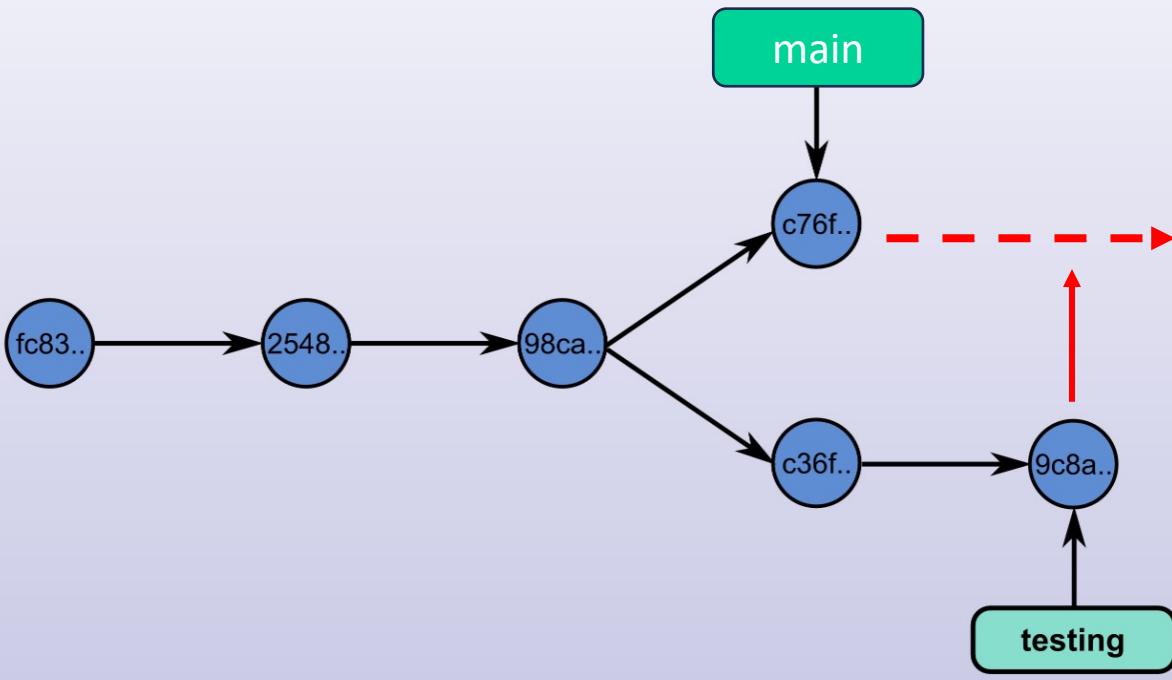
And commit again...



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 *loop.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 *testing* as follows

git branch -d testing

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 "In main branch" > collaborate.txt  
git add collaborate.txt  
git commit -m "Initial commit"
```

- } Start your work in main
- } Working on a branch. This could be work being done by another developer.
- } Back in main work goes on...

```
git checkout -b feature_branch
```

← Create a feature branch

```
echo "different content in branch" > collaborate.txt  
git add collaborate.txt  
git commit -m "Add feature content"
```

```
git checkout main  
echo "Continue work in main branch" >> collaborate.txt  
git add collaborate.txt  
git commit -m "Modify main content"
```

```
git merge feature_branch
```

← Conflicts!

Resolve Conflicts

Run

git status

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

vim collaborate.txt

Decide what changes to keep. Then:

git add collaborate.txt

git commit -m "Resolve conflicts in collaborate.txt"

After you are done merging you should remove the feature branch

git branch -d feature_branch

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