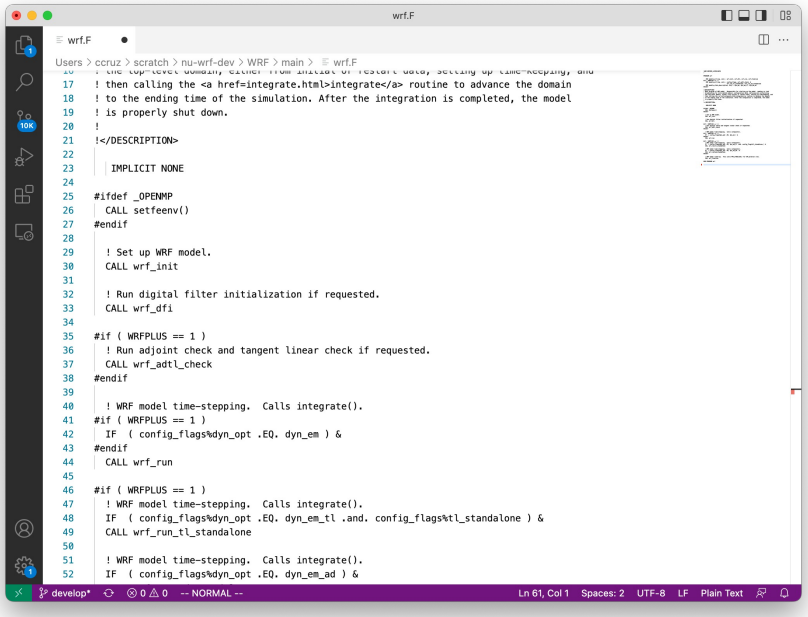
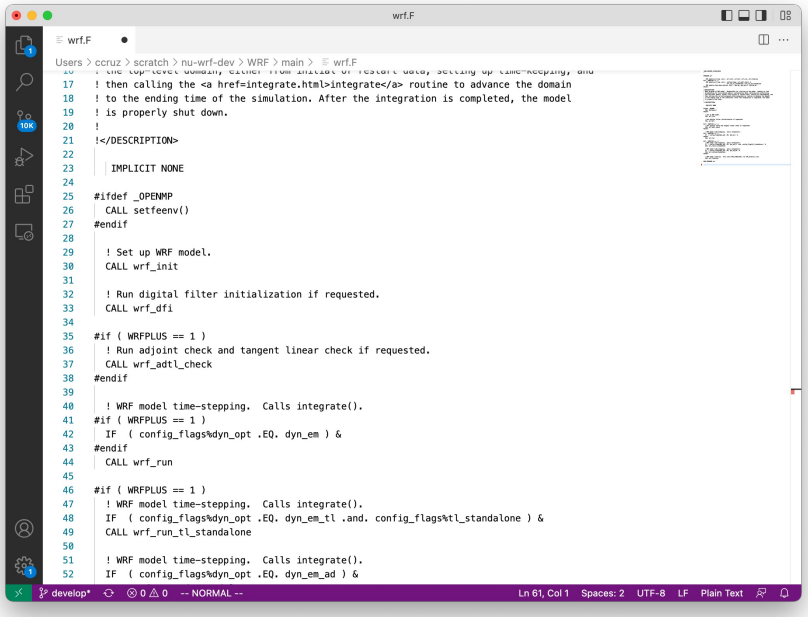
Introduction to Version Control with Git

Fall 2024 ASTG Training

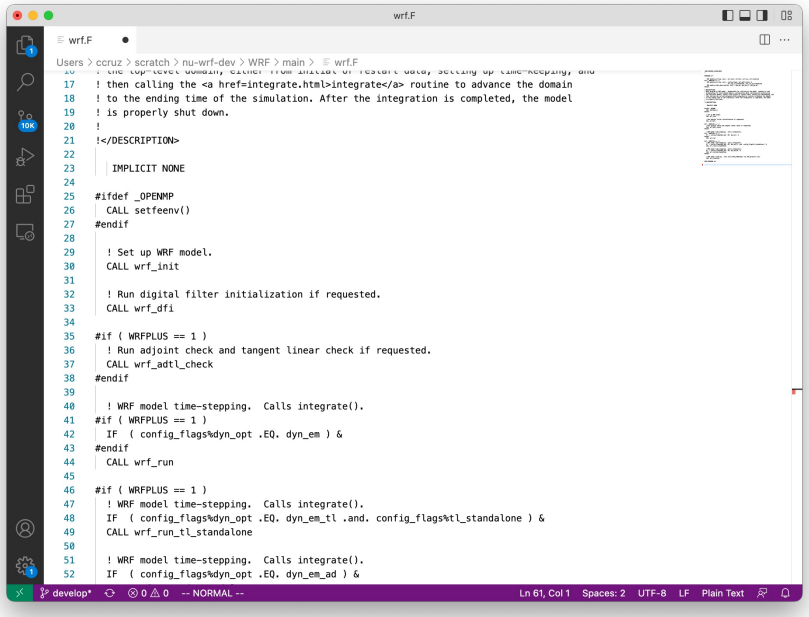
Motivation


wrf.F
Users > ccruz > scratch > nu-wrf-dev > WRF > main > wrf.F
! then calling the integrate 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
#ifdef _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 .EQ. dyn_em) &
#endif
CALL wrf_run
#if (WRFPLUS == 1)
! WRF model time-stepping. Calls integrate().
IF (config_flags%dyn_opt .EQ. dyn_em_tl .and. config_flags%tl_standalone) &
CALL wrf_run_tl_standalone
! WRF model time-stepping. Calls integrate().
IF (config_flags%dyn_opt .EQ. dyn_em_ad) &
develop*
-- NORMAL --
Ln 61, Col 1
Spaces: 2
UTF-8
LF
Plain Text

Motivation

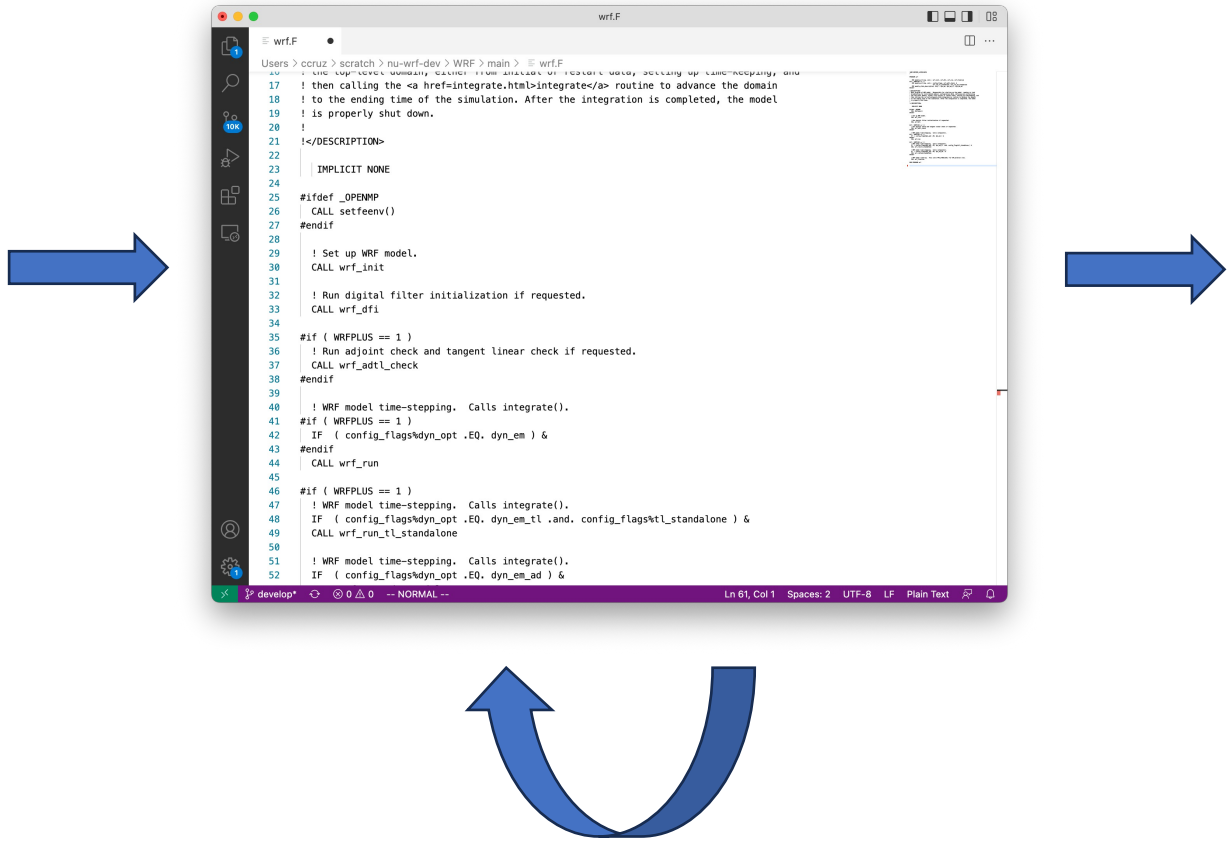

wrf.F
Users > ccruz > scratch > nu-wrf-dev > WRF > main > wrf.F
! then calling the integrate routine to advance the domain
! to the ending time of the simulation. After the integration is completed, the model
! is properly shut down.
!
!</DESCRIPTION>
IMPLICIT NONE
#ifdef _OPENMP
CALL setfeenv()
#endif
! 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 .EQ. dyn_em) &
#endif
CALL wrf_run
#if (WRFPLUS == 1)
! WRF model time-stepping. Calls integrate().
IF (config_flags%dyn_opt .EQ. dyn_em_tl .and. config_flags%tl_standalone) &
CALL wrf_run_tl_standalone
! WRF model time-stepping. Calls integrate().
IF (config_flags%dyn_opt .EQ. dyn_em_ad) &
develop*
-- NORMAL --
Ln 61, Col 1
Spaces: 2
UTF-8
LF
Plain Text

Motivation


wrf.F
Users > ccruz > scratch > nu-wrf-dev > WRF > main > wrf.F
! then calling the integrate 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
#ifdef _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 .EQ. dyn_em) &
#endif
CALL wrf_run
#if (WRFPLUS == 1)
! WRF model time-stepping. Calls integrate().
IF (config_flags%dyn_opt .EQ. dyn_em_tl .and. config_flags%tl_standalone) &
CALL wrf_run_tl_standalone
! WRF model time-stepping. Calls integrate().
IF (config_flags%dyn_opt .EQ. dyn_em_ad) &
develop*
-- NORMAL --
Ln 61, Col 1
Spaces: 2
UTF-8
LF
Plain Text

Software development includes the continuous process of modifying programs

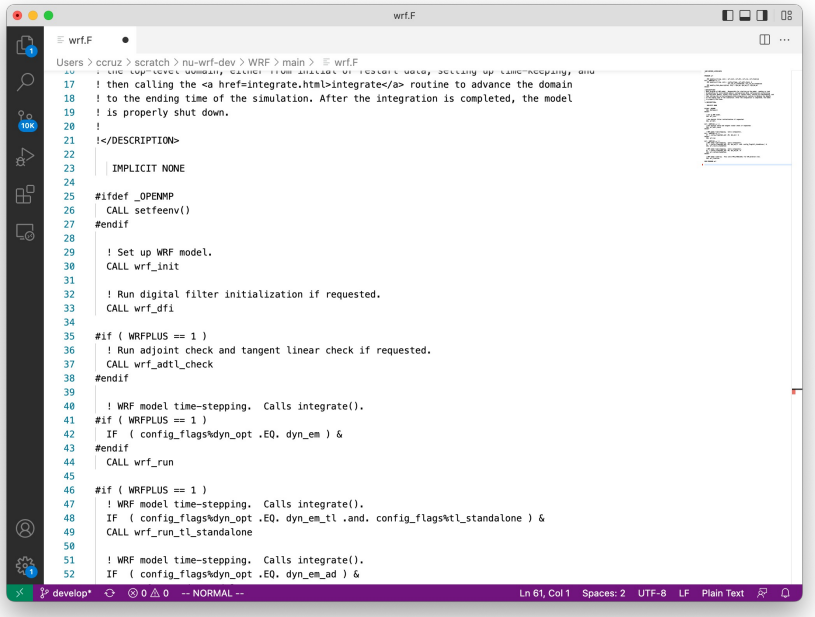
Proper code management is essential for effective and sustainable software development



Software development includes the continuous process of modifying programs

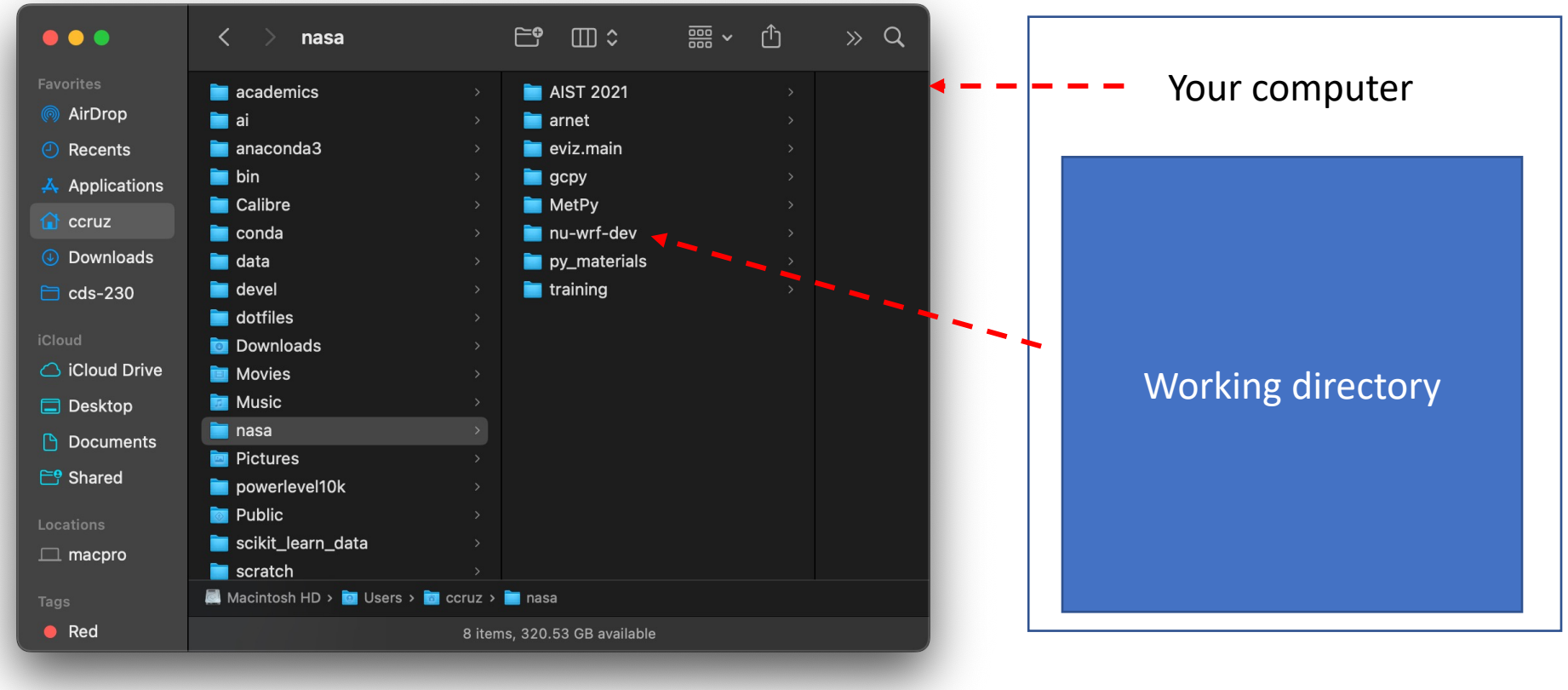
Proper code management is essential for effective and sustainable software development

Requires a version control system (**VCS**[1])



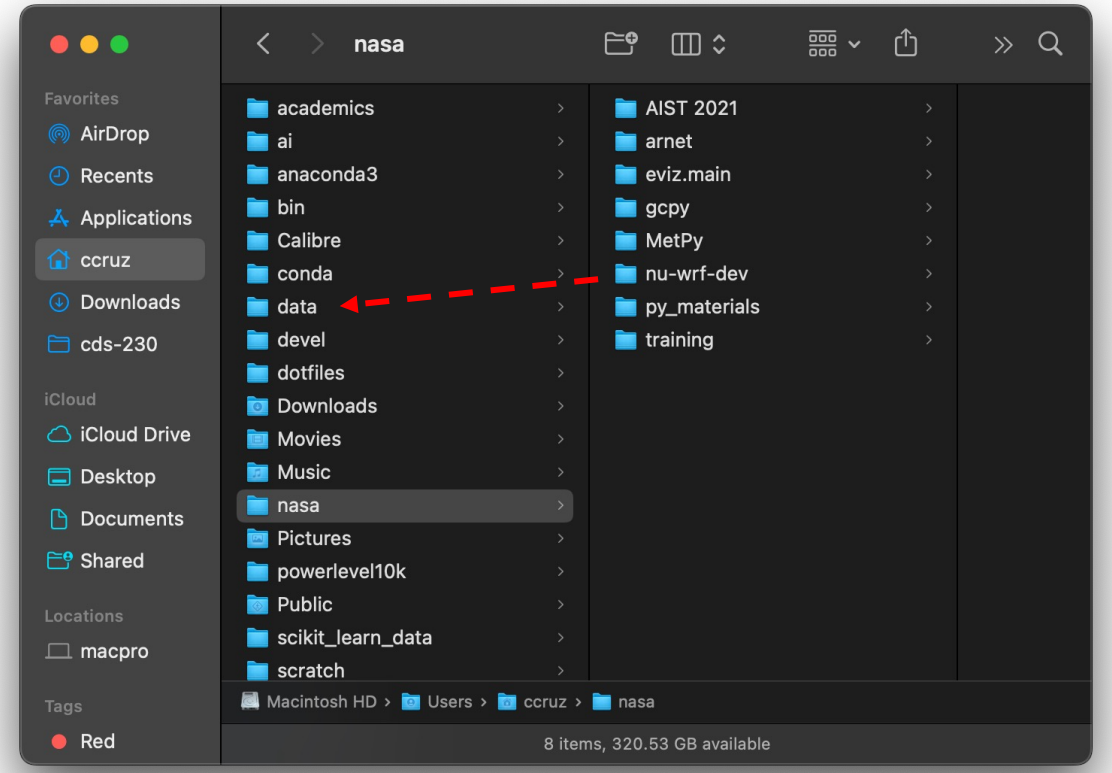
[1]Another technical term is Source Code Management (SCM)

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



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

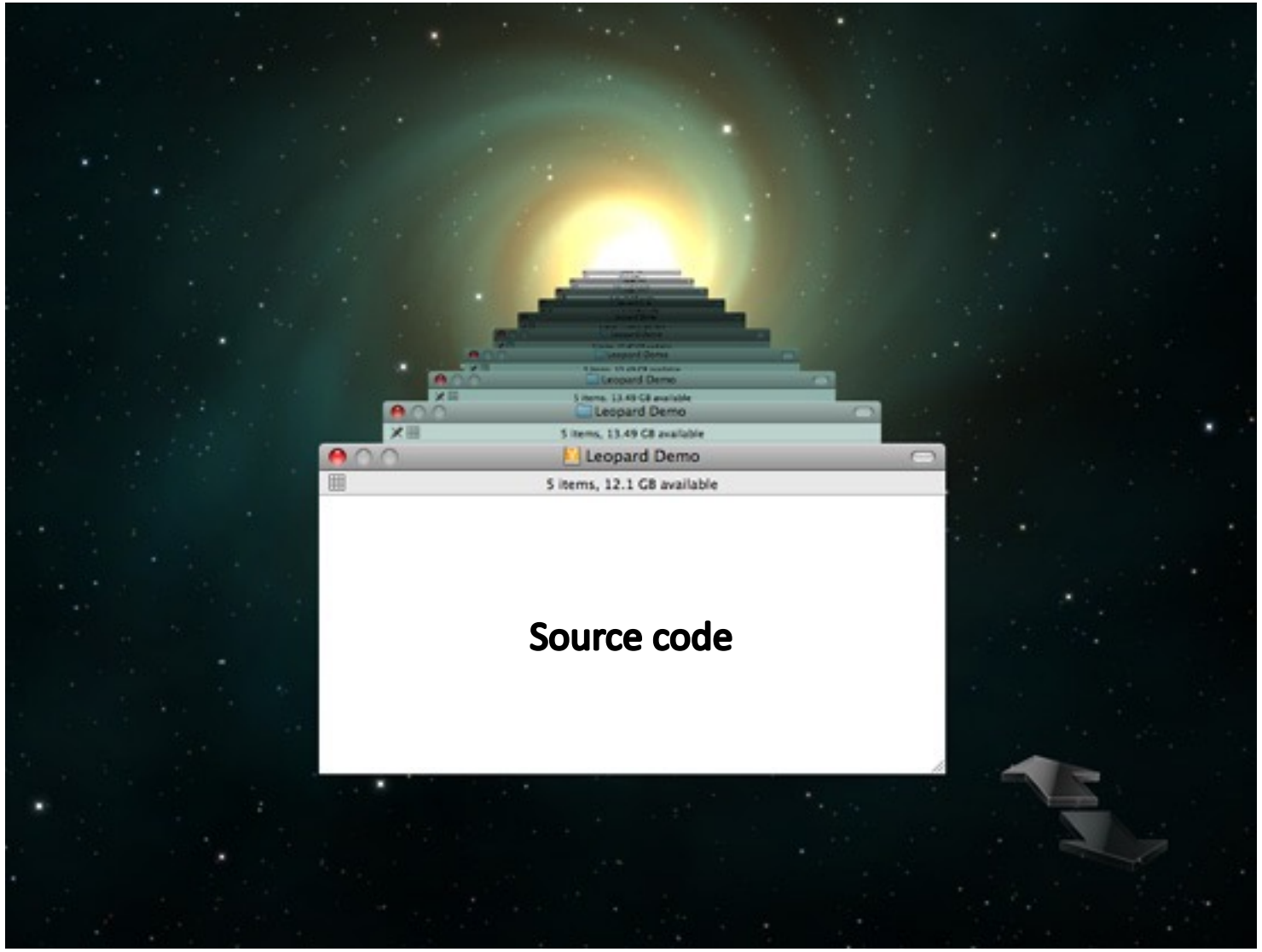
A personal “version control system”



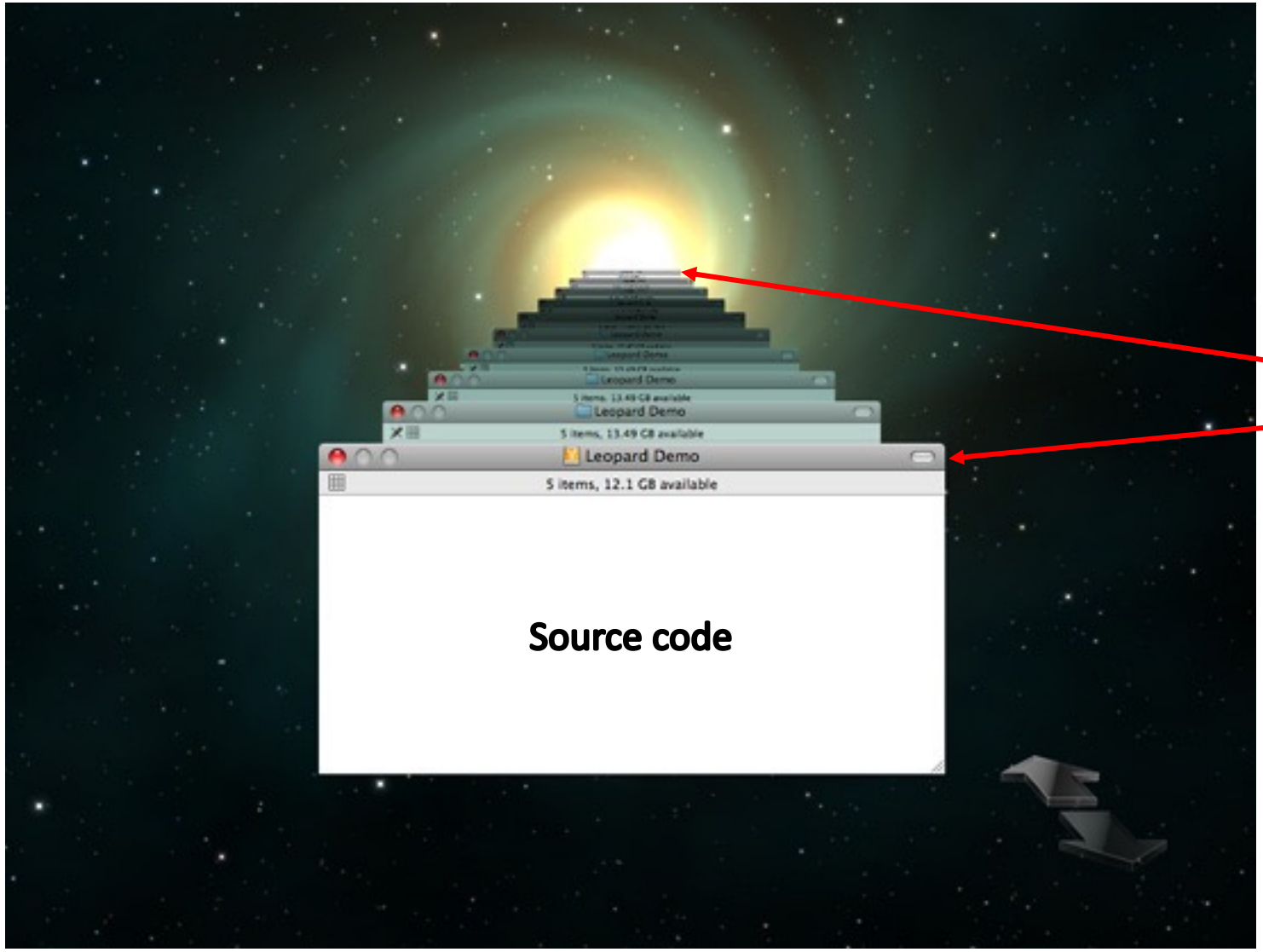
Your computer

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

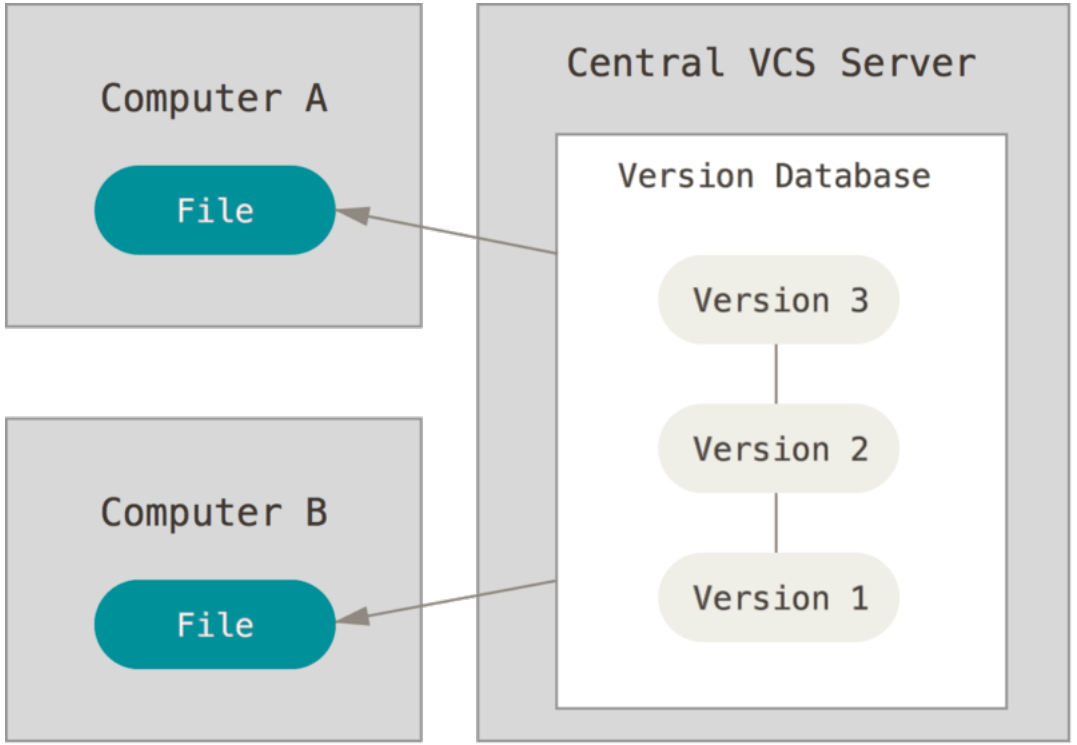
Version Control System (VCS)



Version Control System (VCS)



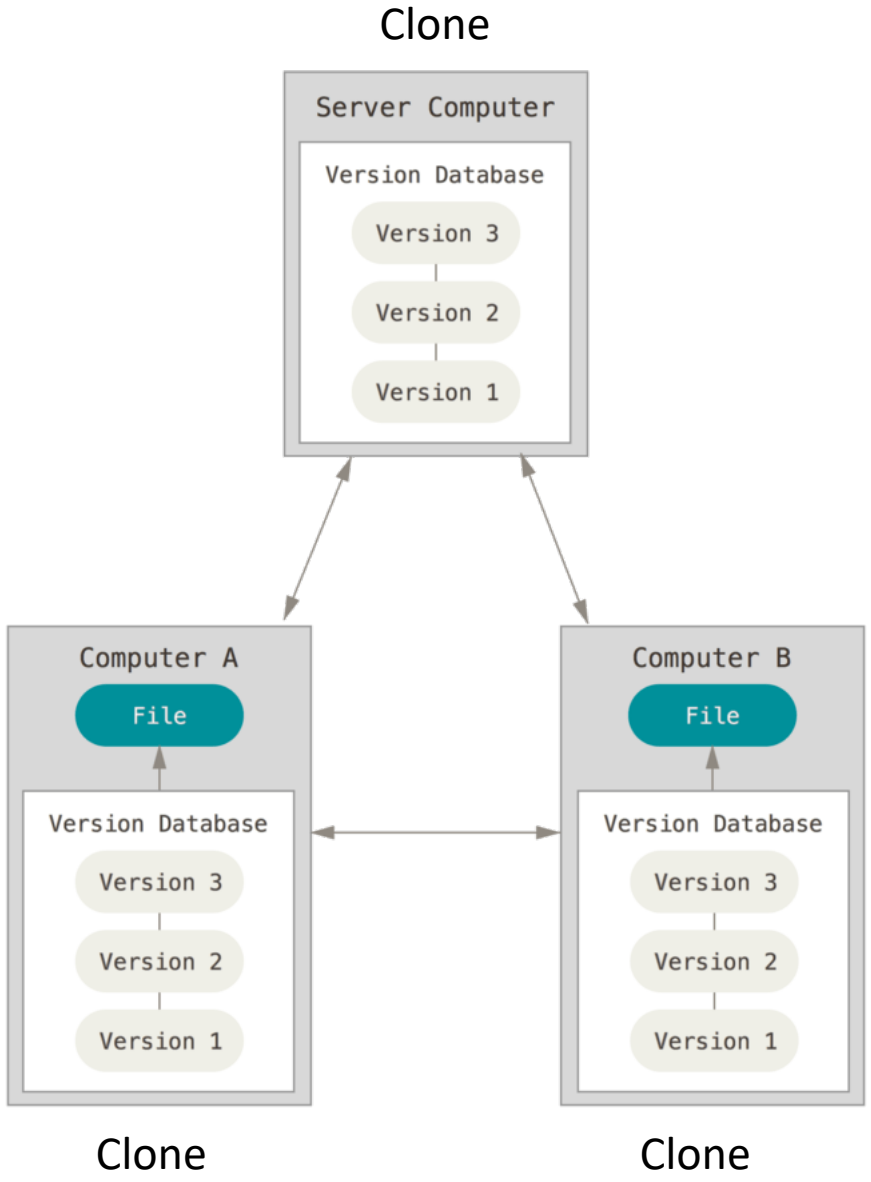
Centralized VCS



Examples:

RCS (c. 1982)
CVS (c. 1990)
Subversion (c. 2000)

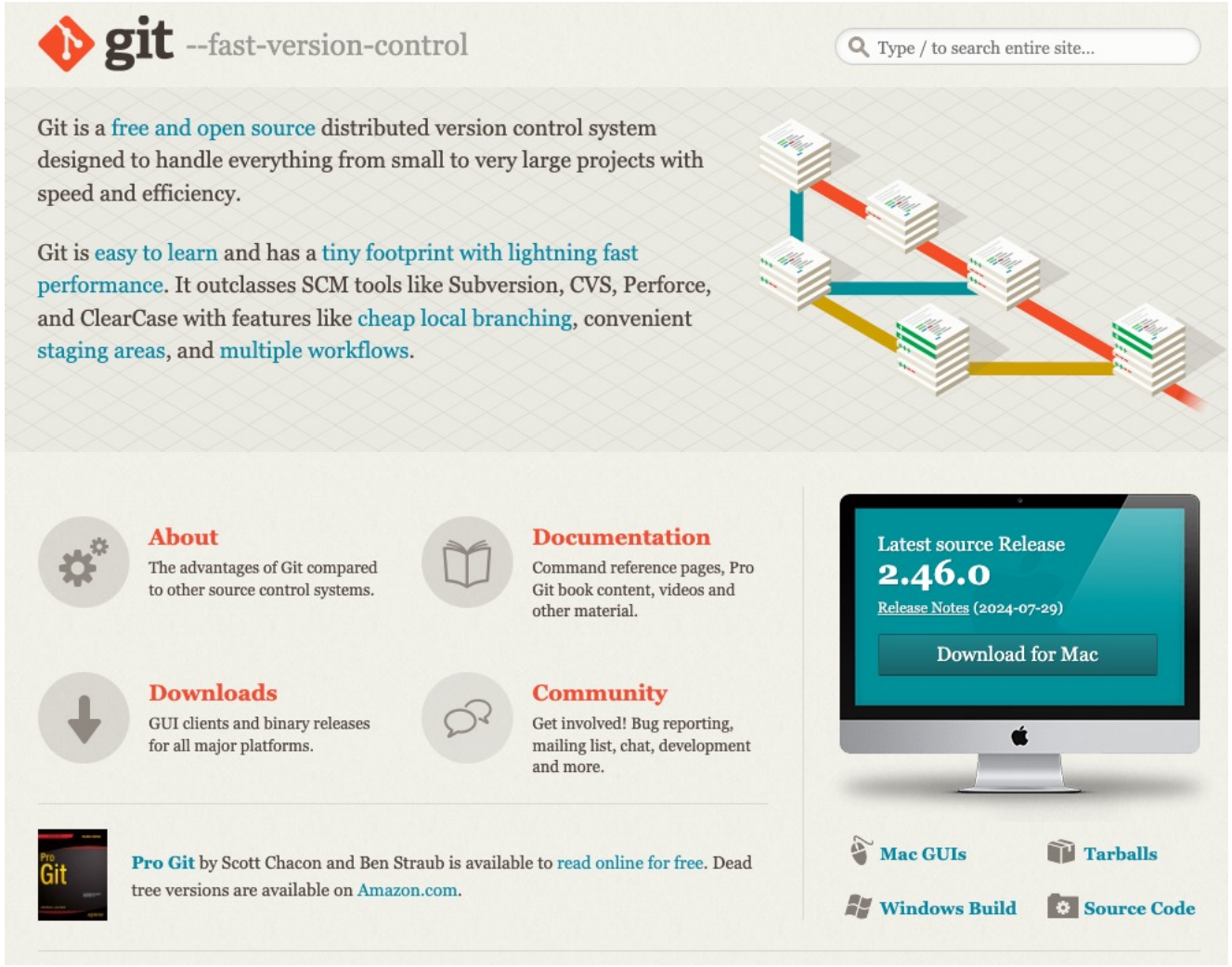
Distributed VCS



Examples:

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

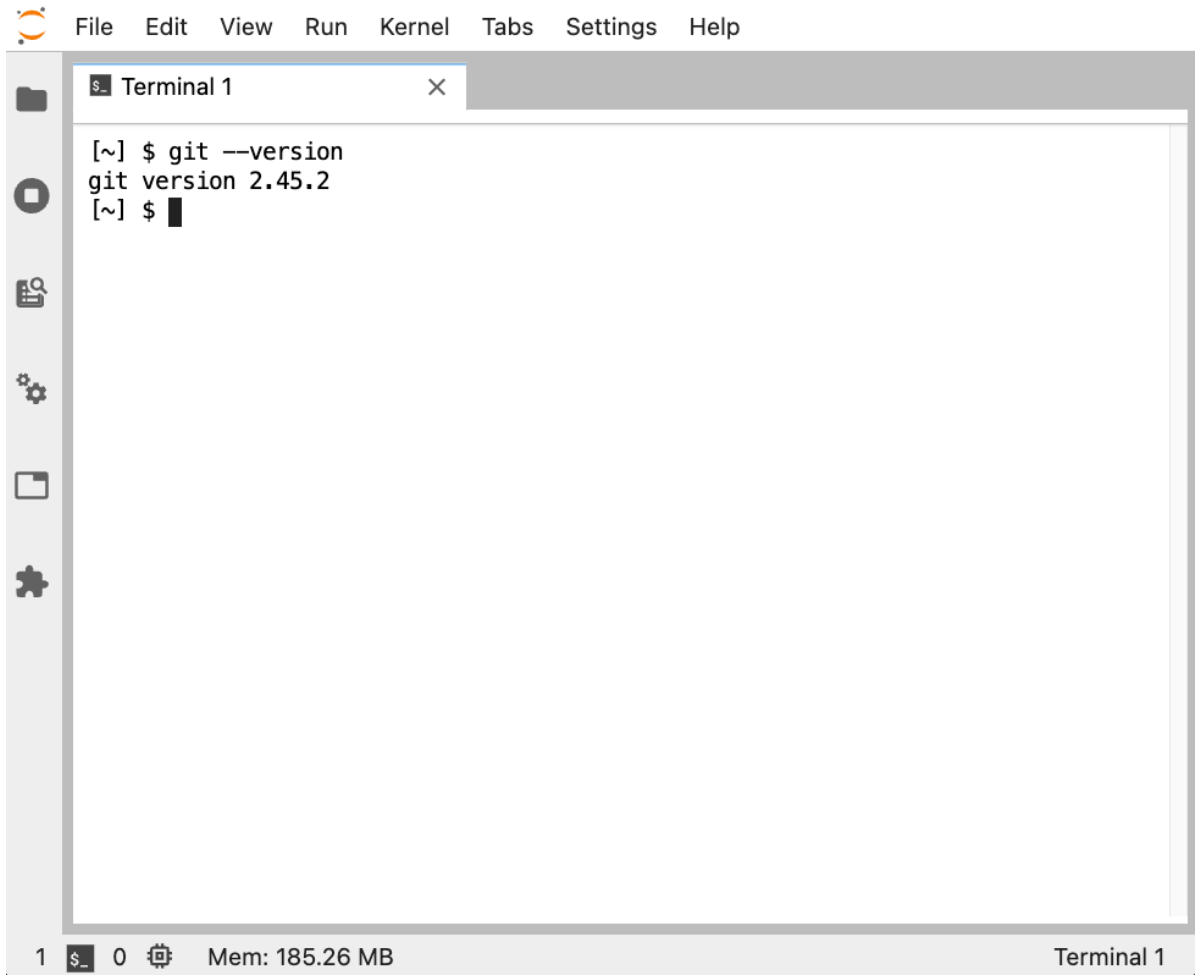
Git is an implementation of a distributed VCS



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



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

Using Git

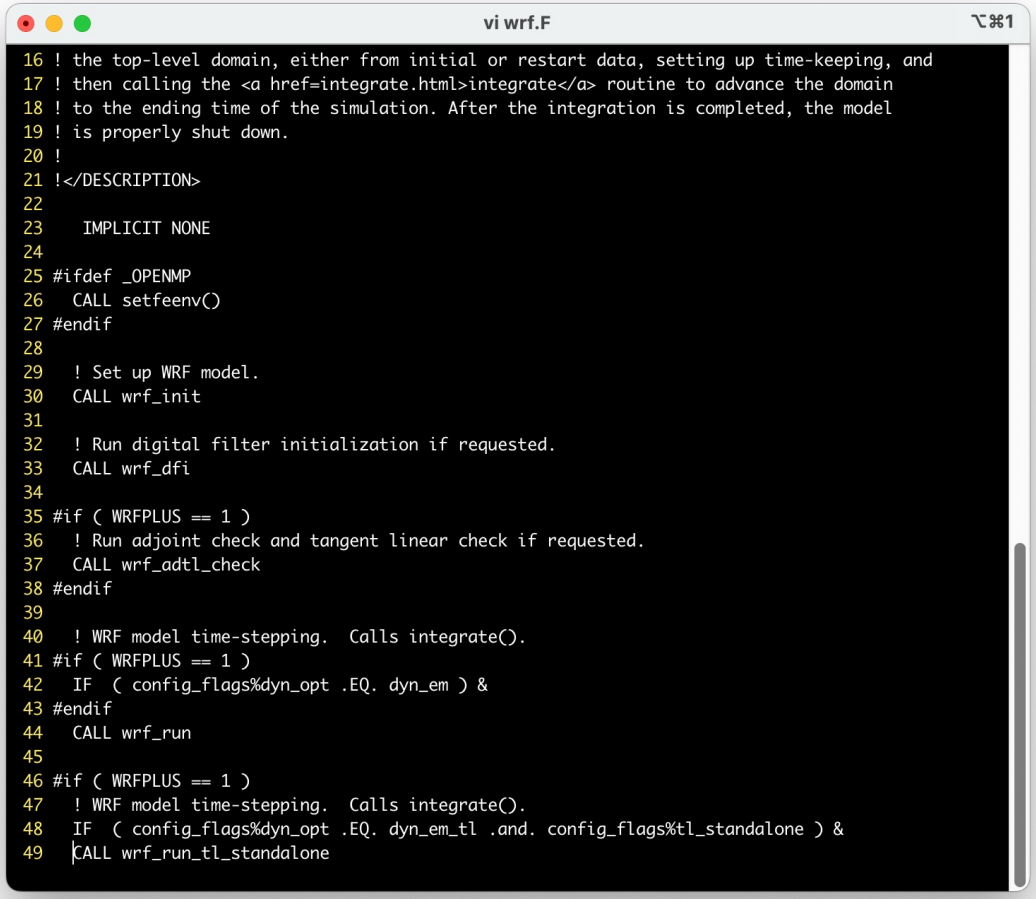
- You may use your laptop if it has already Git installed

- If you need to install Git, you can do so later (start) here

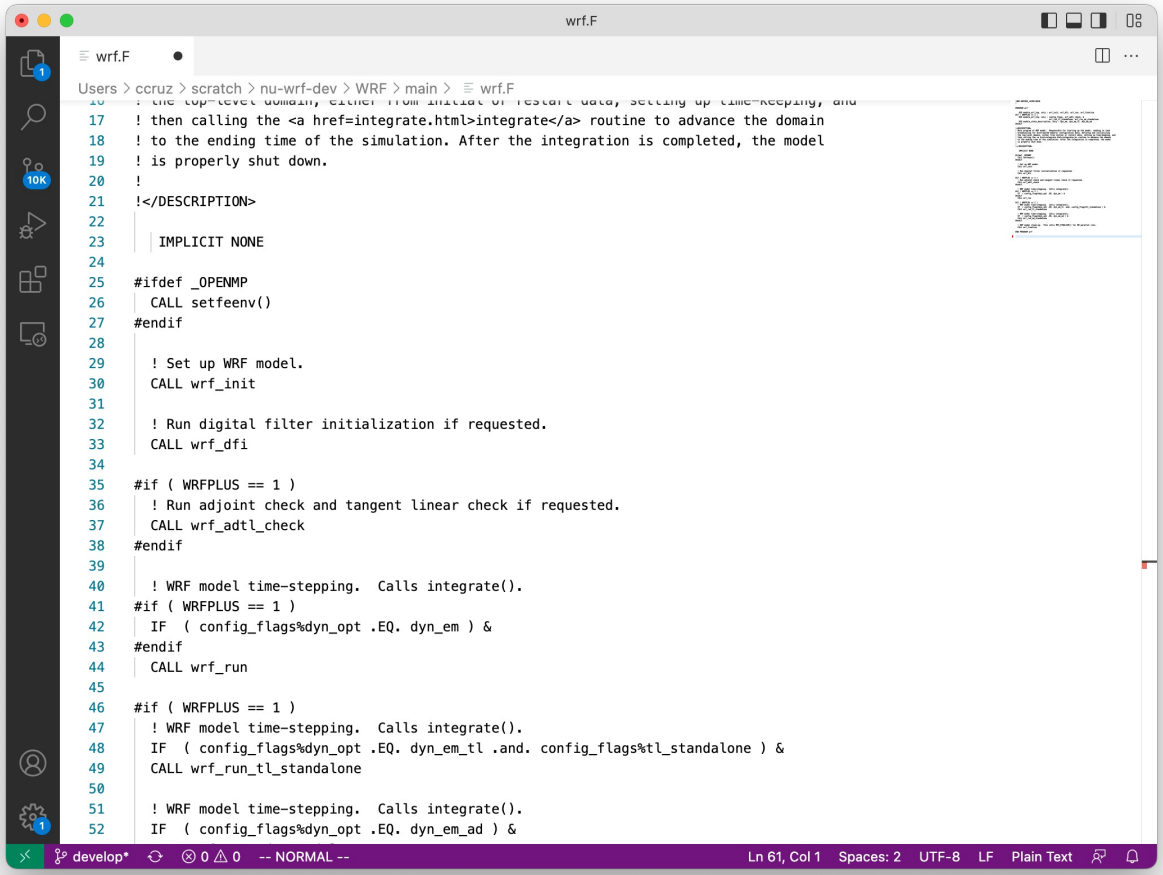
Official website:
https://git-scm.com/

Creating and modifying source code

Code Editors: vim, emacs, nano, etc.
IDEs: VS Code, PyCharm, Sublime*, etc.



vim



Visual Studio

Configuring Git

- **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:

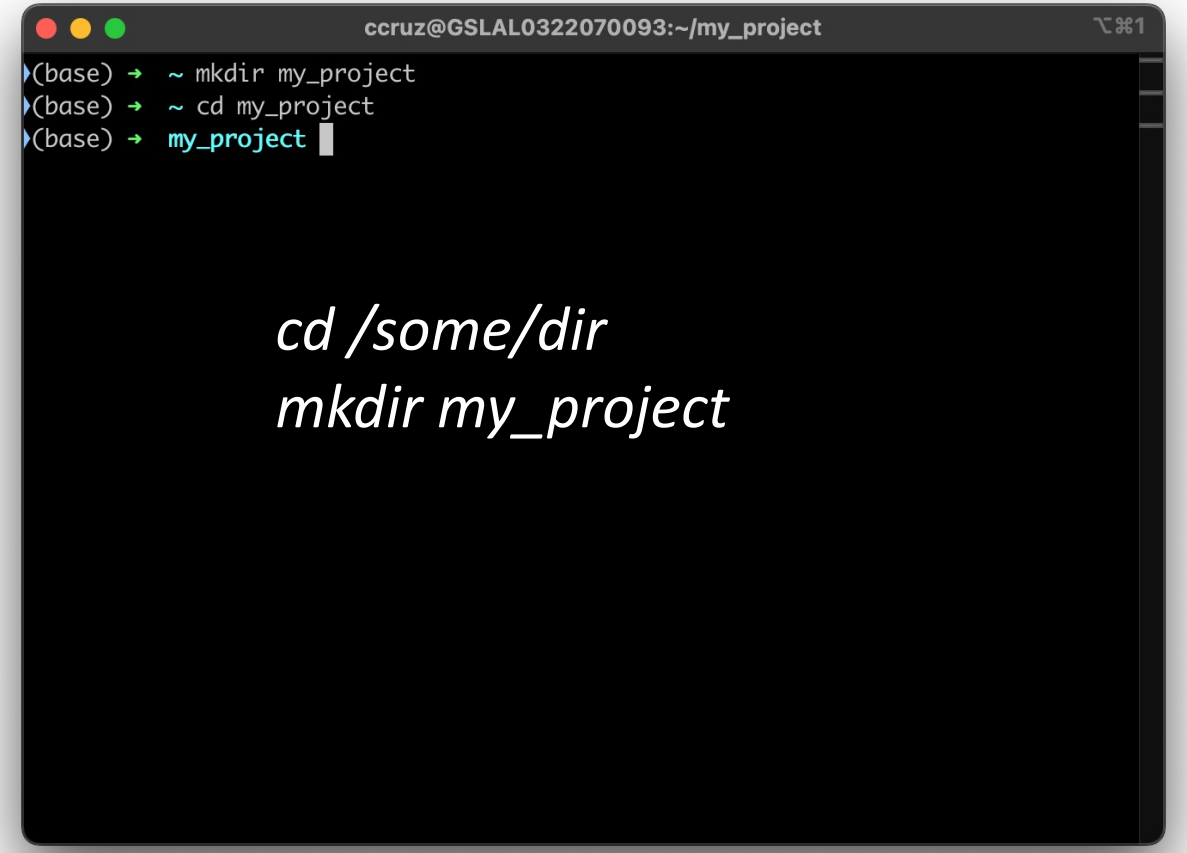
```
[user]
        name = Your Name Goes Here
        email = yourusername@domain.com
[core]
        editor = /usr/bin/vim
```

Check your settings by running:

$ git config --list

Create a Working Directory*

Exercise
run the following commands on a terminal:

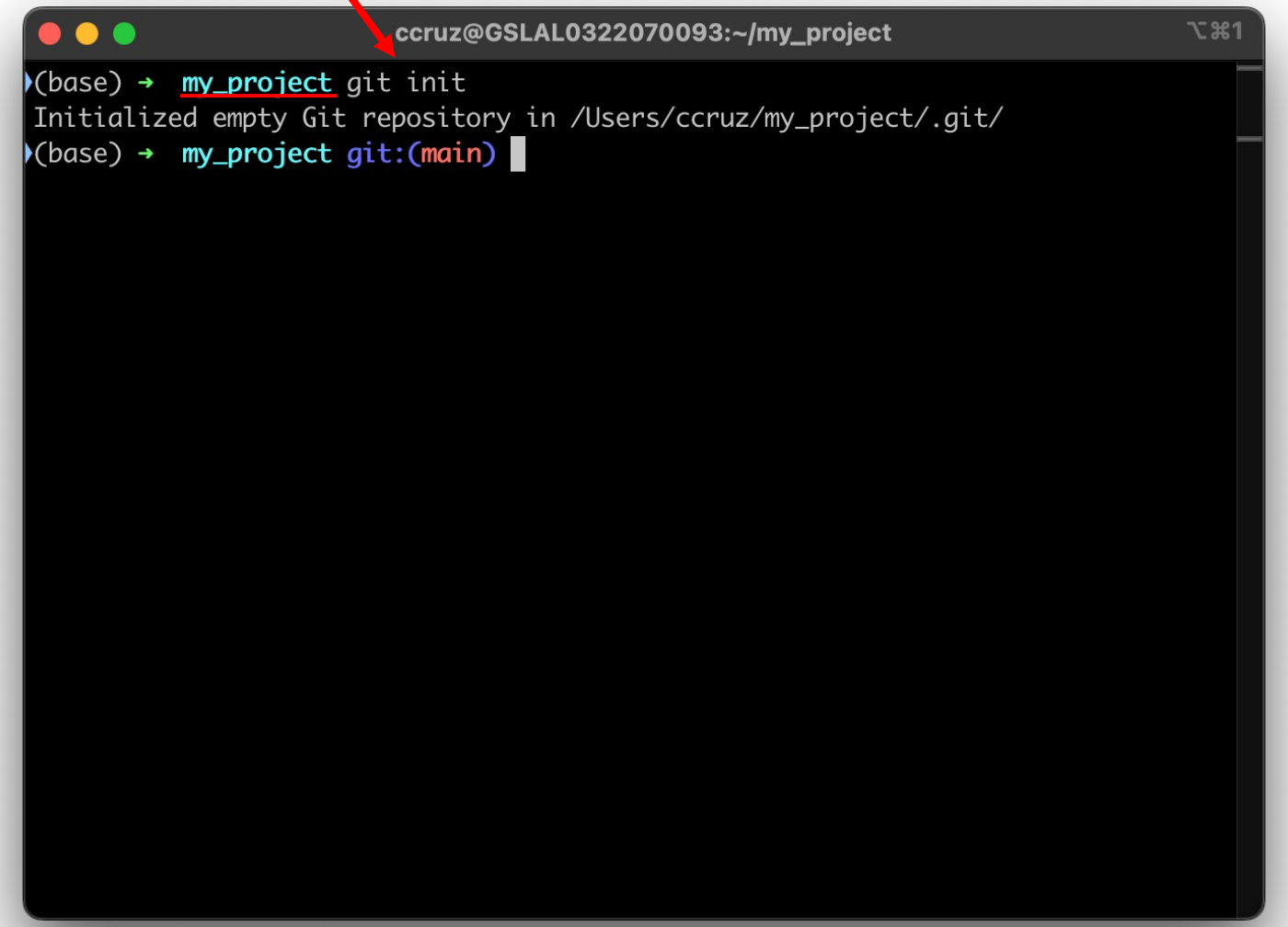


For this tutorial, go to some directory on your computer and create a working directory called ***my_project.***

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

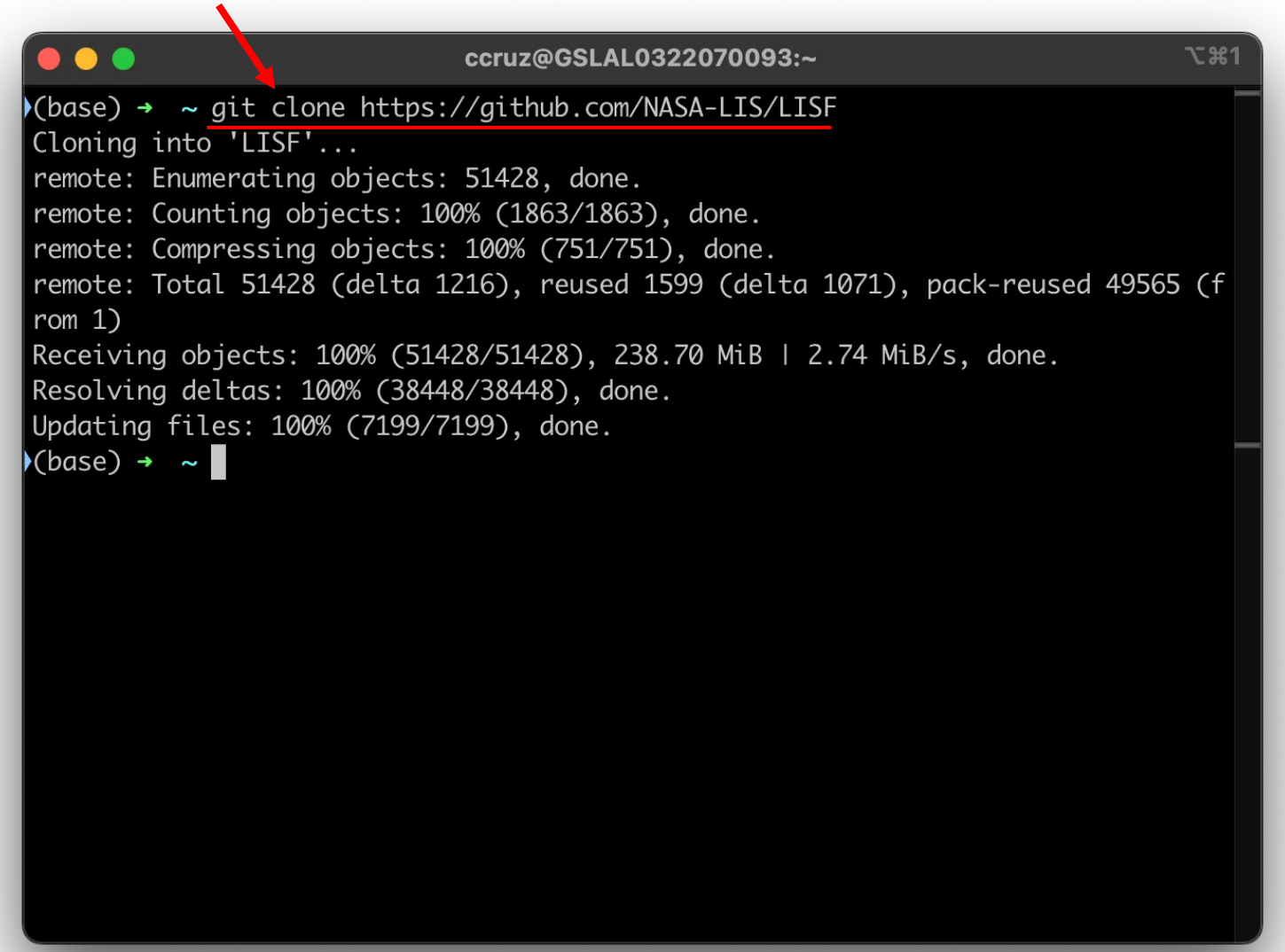
Creating a “repo”

$ git init



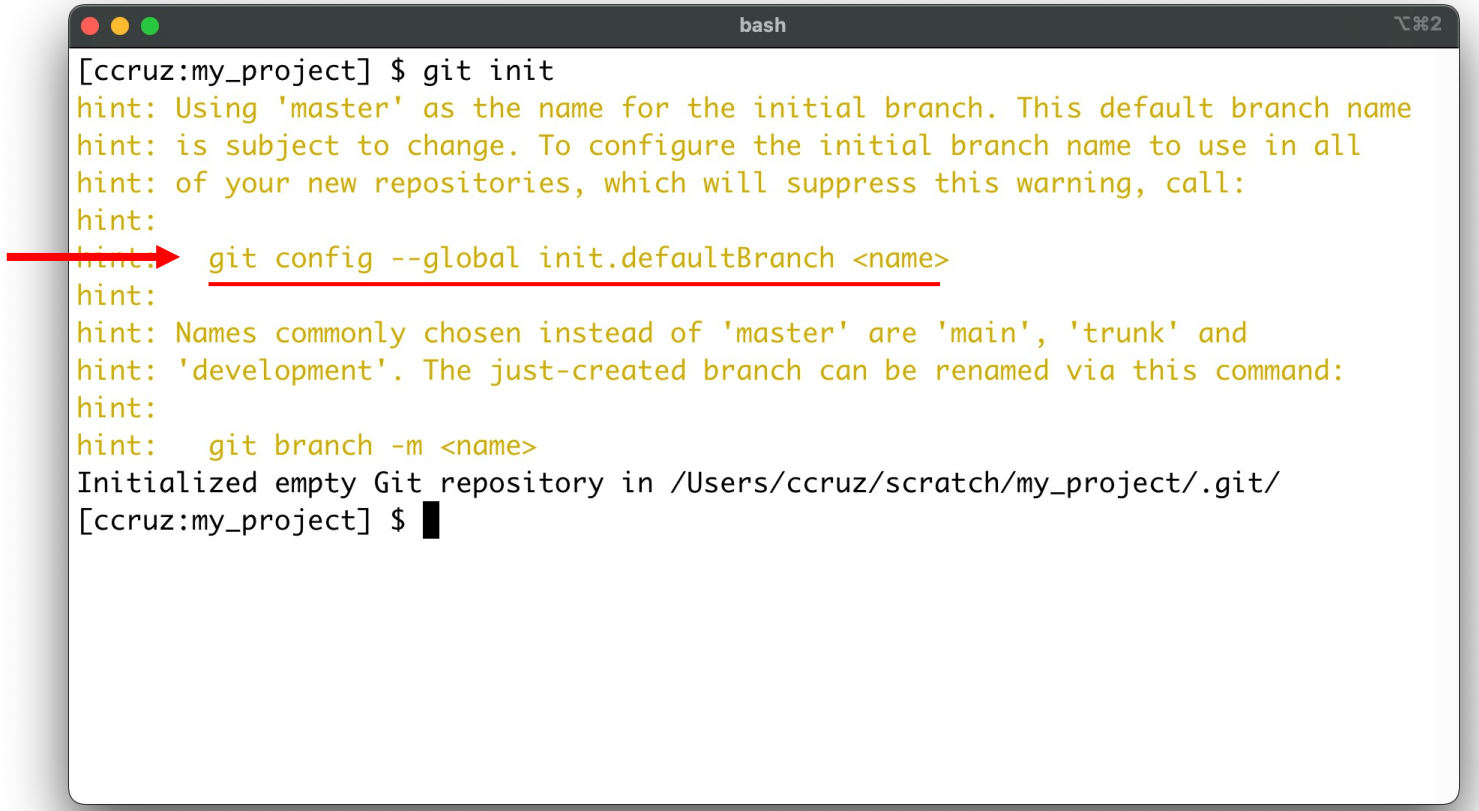
Creating a “repo”

$ git clone



Creating a “repo”

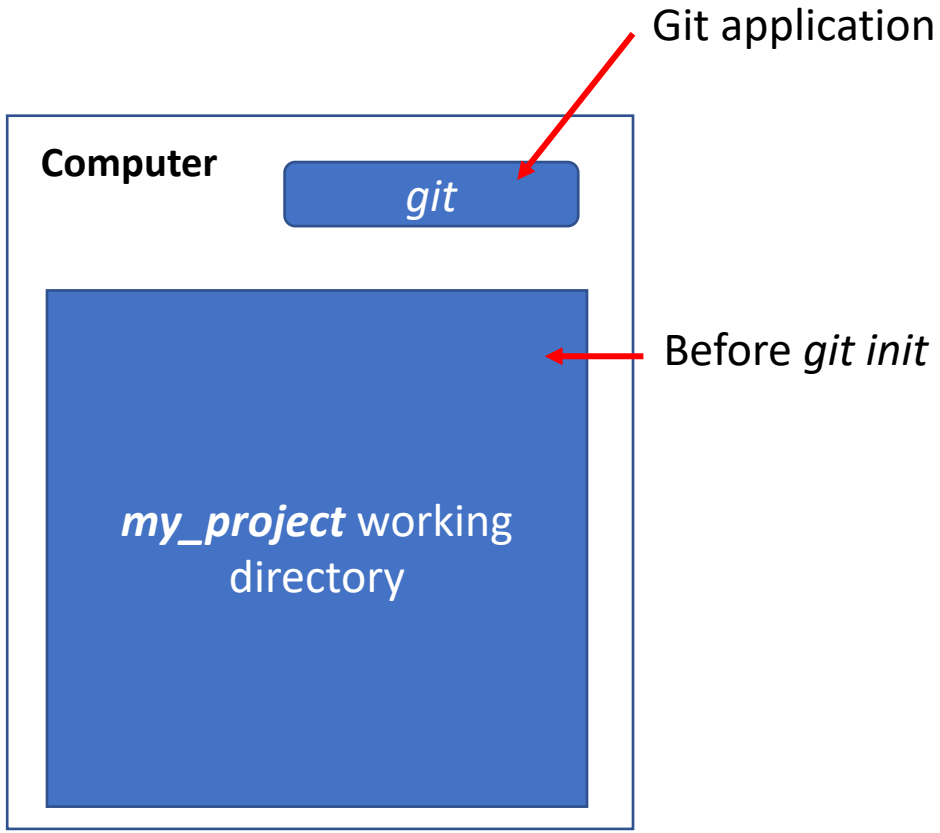
$ *git init*



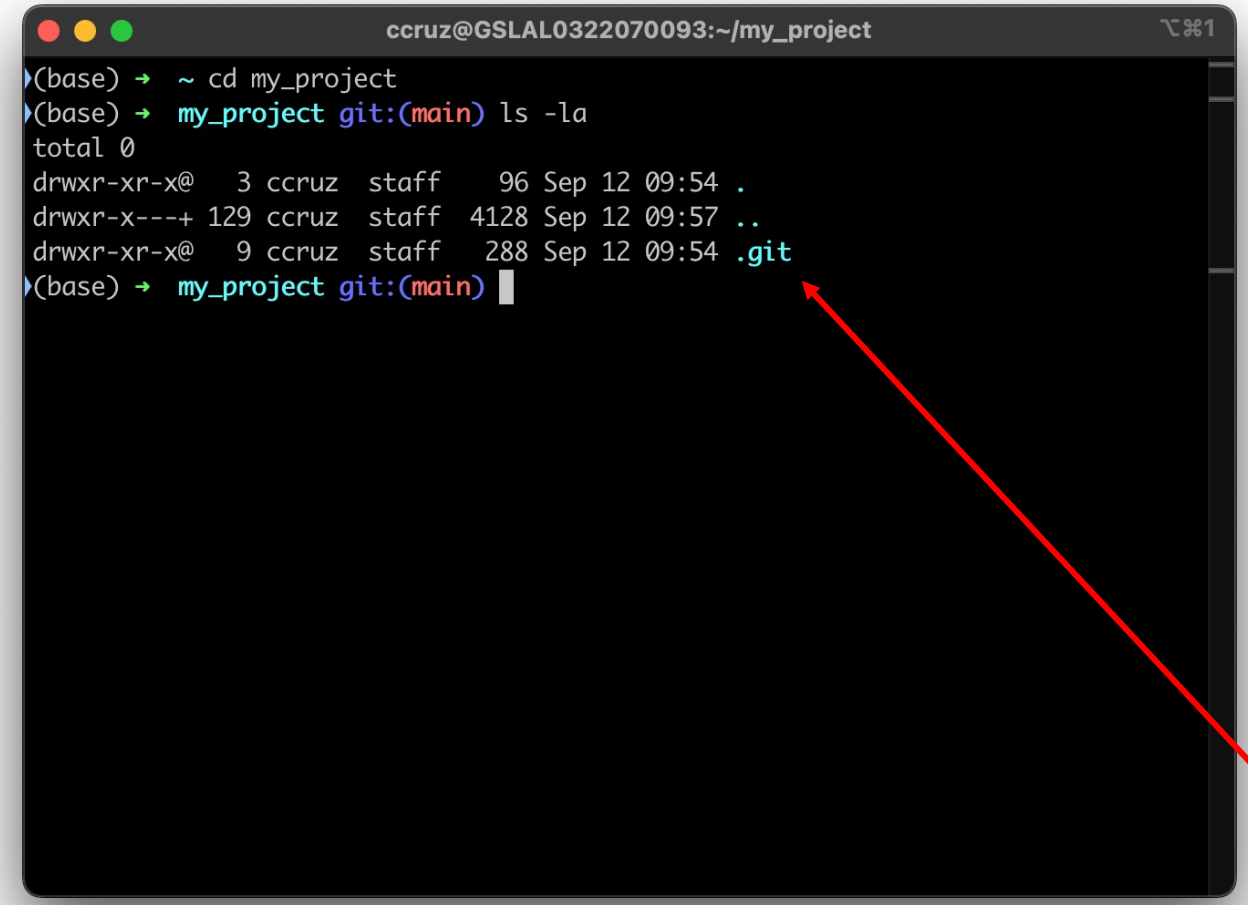
Create an Empty Repository

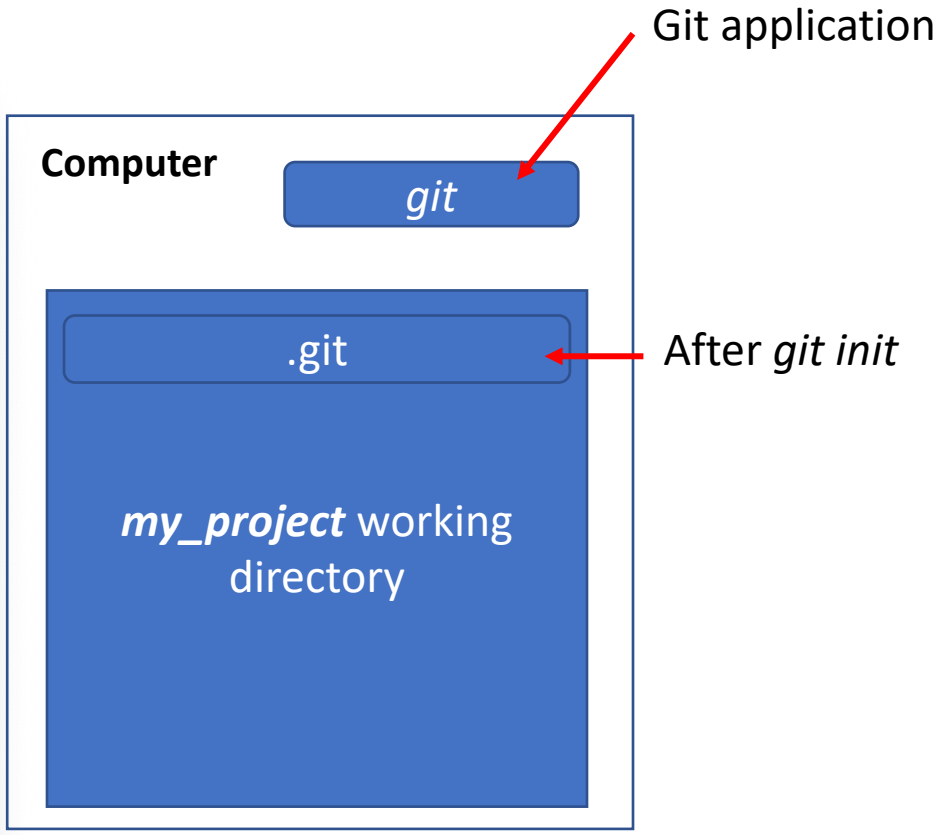
Exercise

run the following commands on your terminal:



Create an Empty Repository





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]

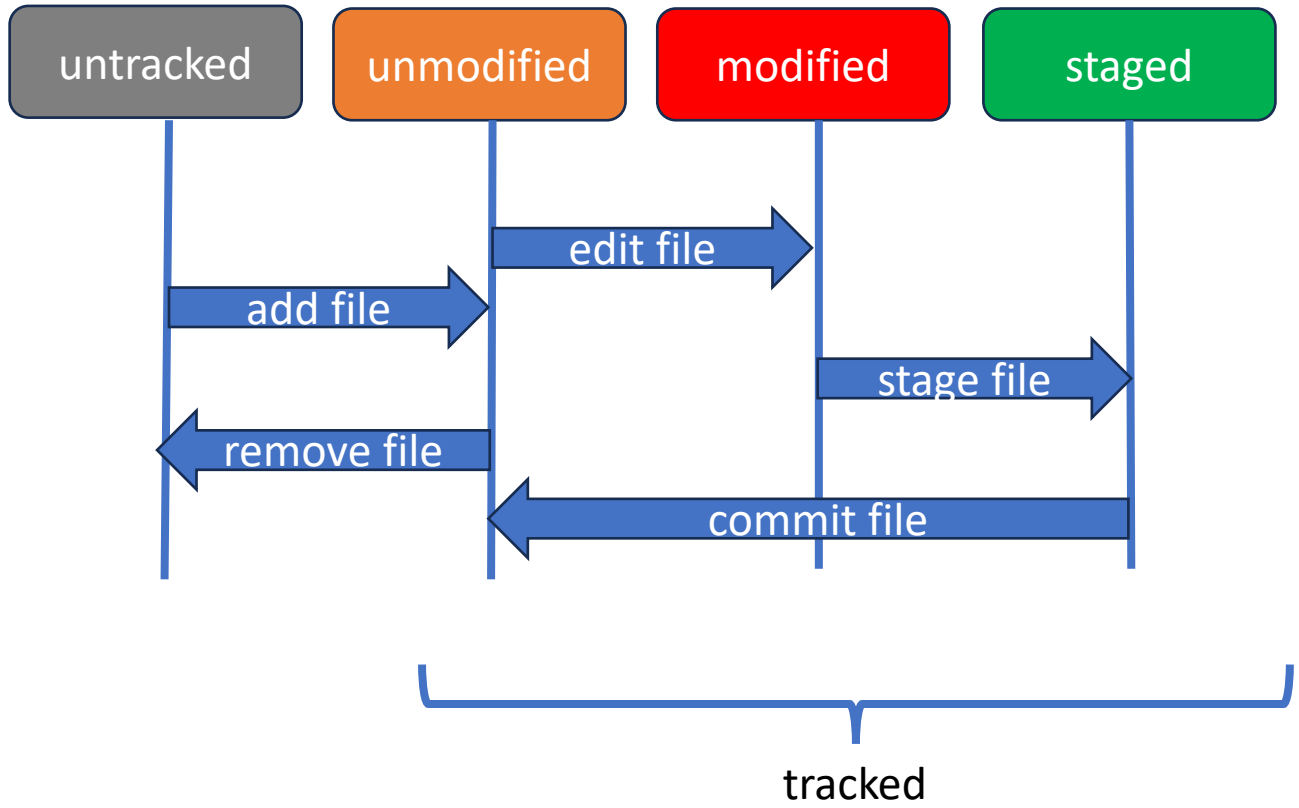
Unstages the file, but preserve its contents

$ 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

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

Check repo status again

git status

Let's check the history

git log

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*

Canceling 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
```

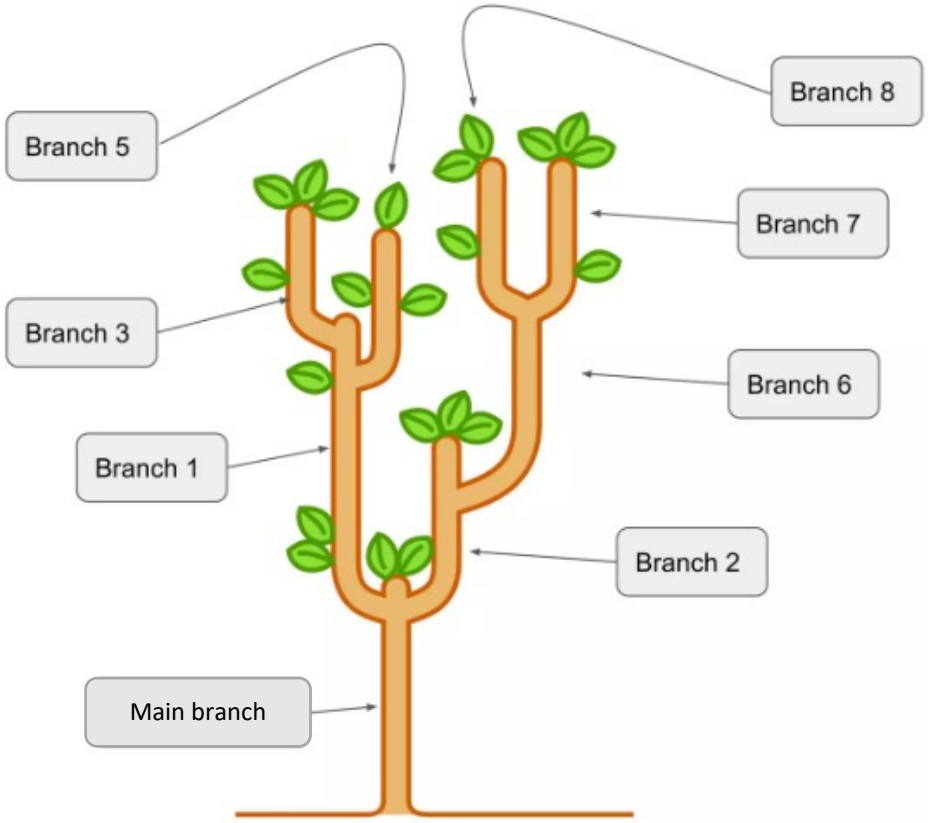
Run the following commands and observe what happens

```
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.

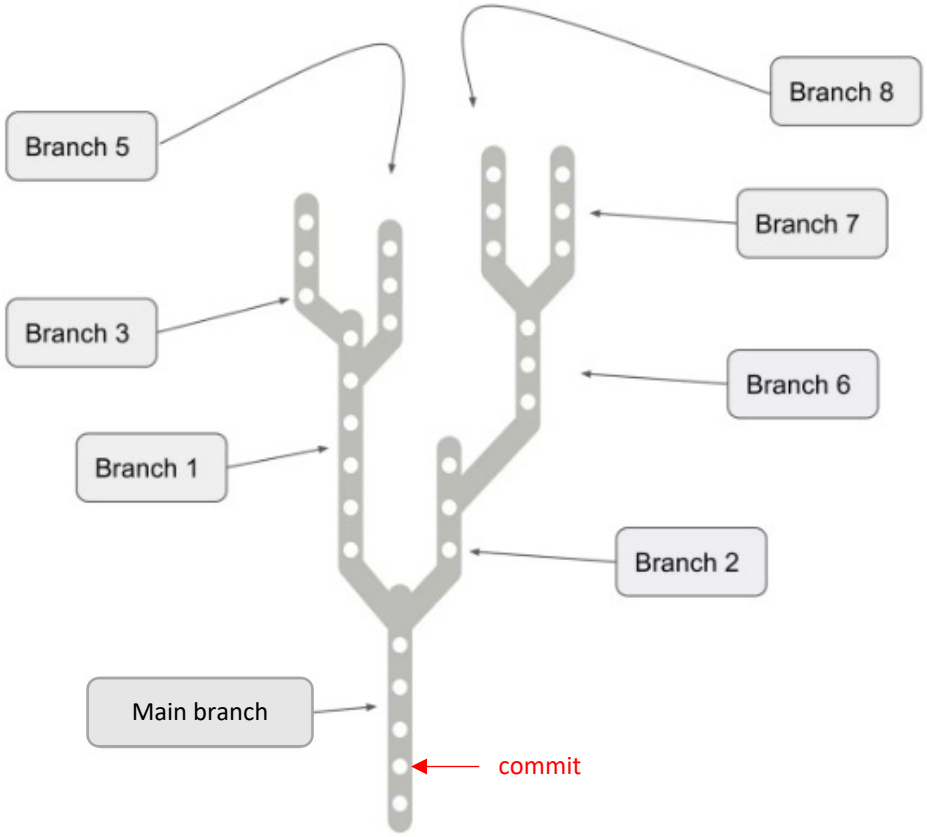
Branches

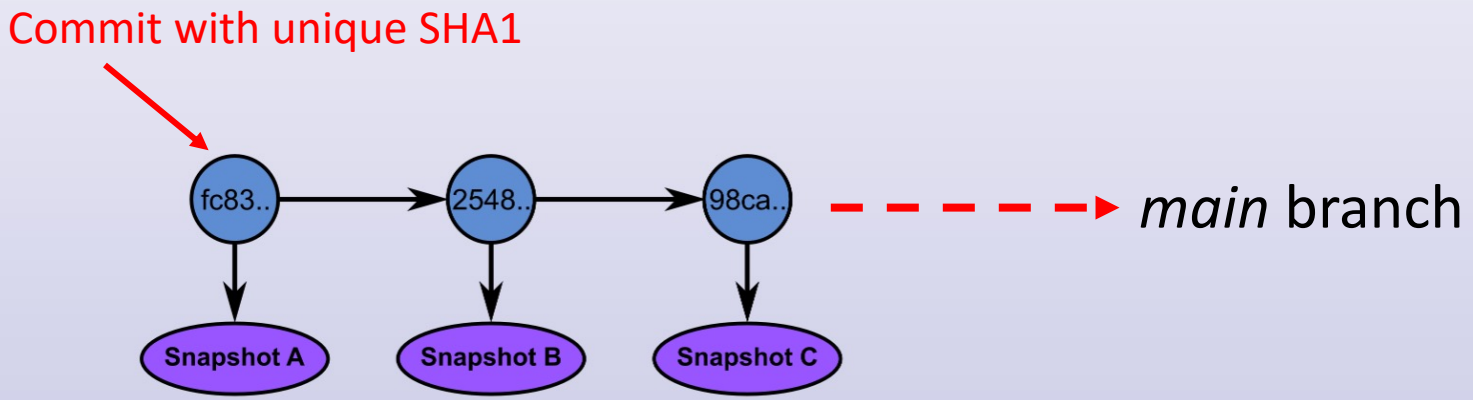
- What is a branch?
- Creating branches (git branch)
- Merging branches (git merge)
- Resolving merge conflicts



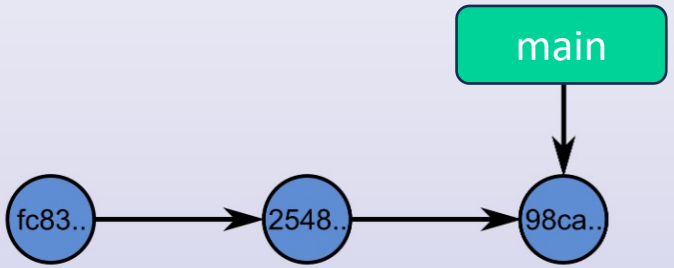
Branches

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



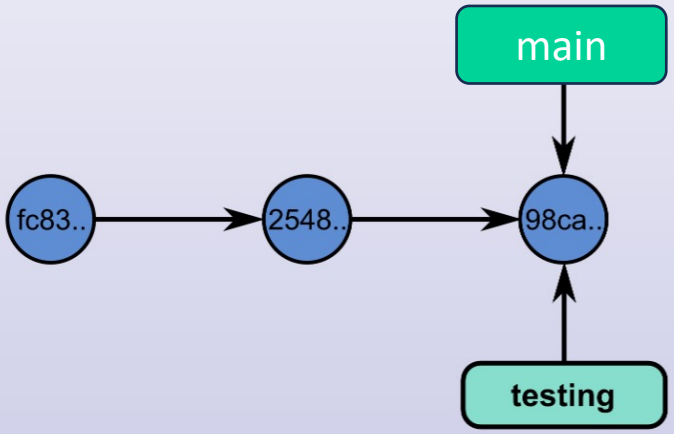


Commits are repository snapshots



main is a branch created during *git init*

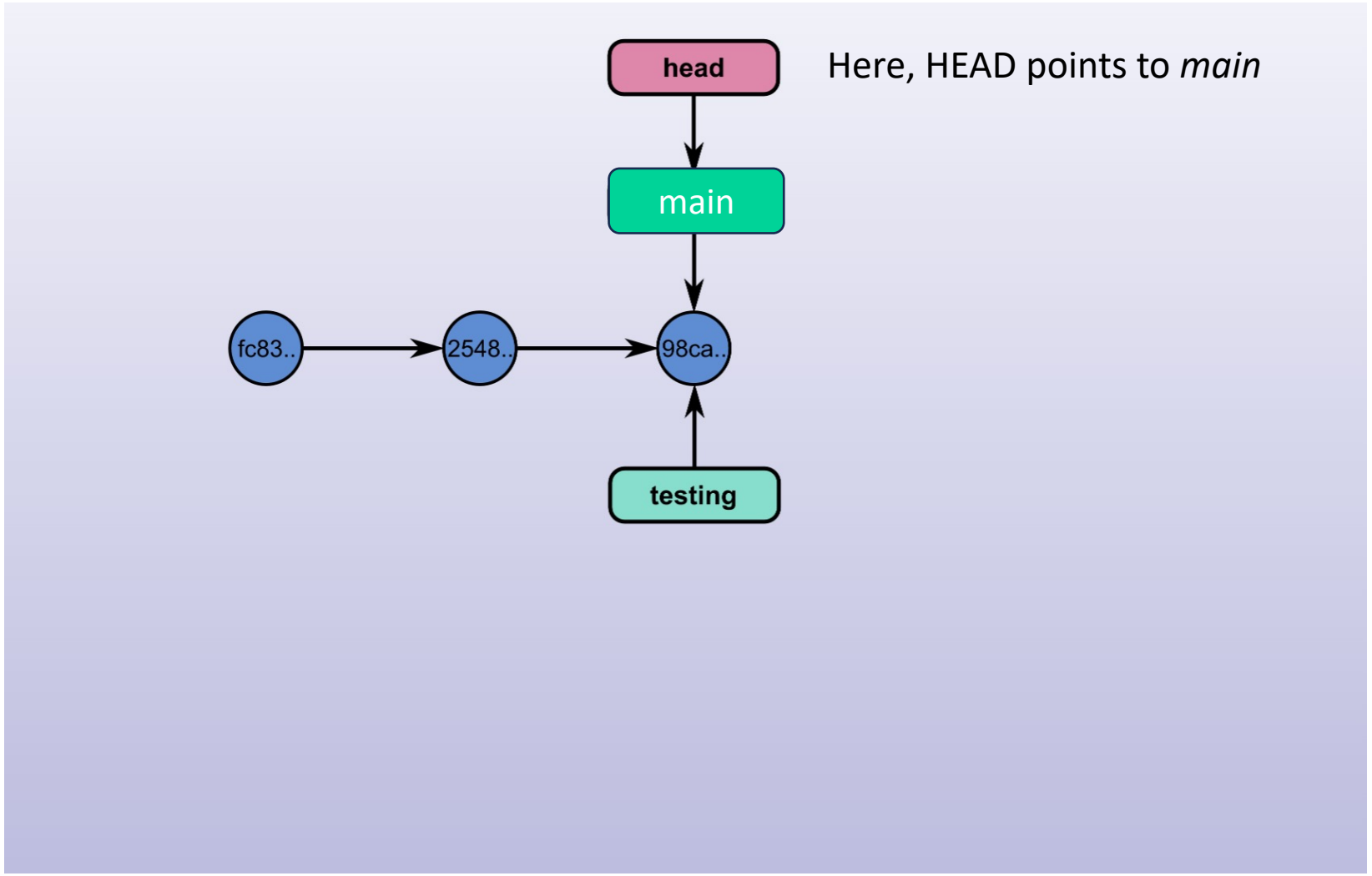
A branch is a pointer to a commit



Create branch

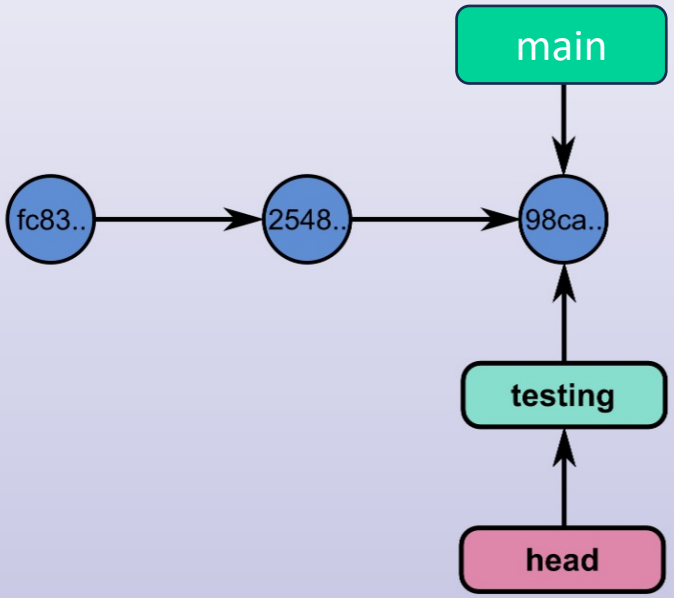
git branch testing

We can have many branches.



Here, HEAD points to *main*

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



HEAD pointer now points to *testing*

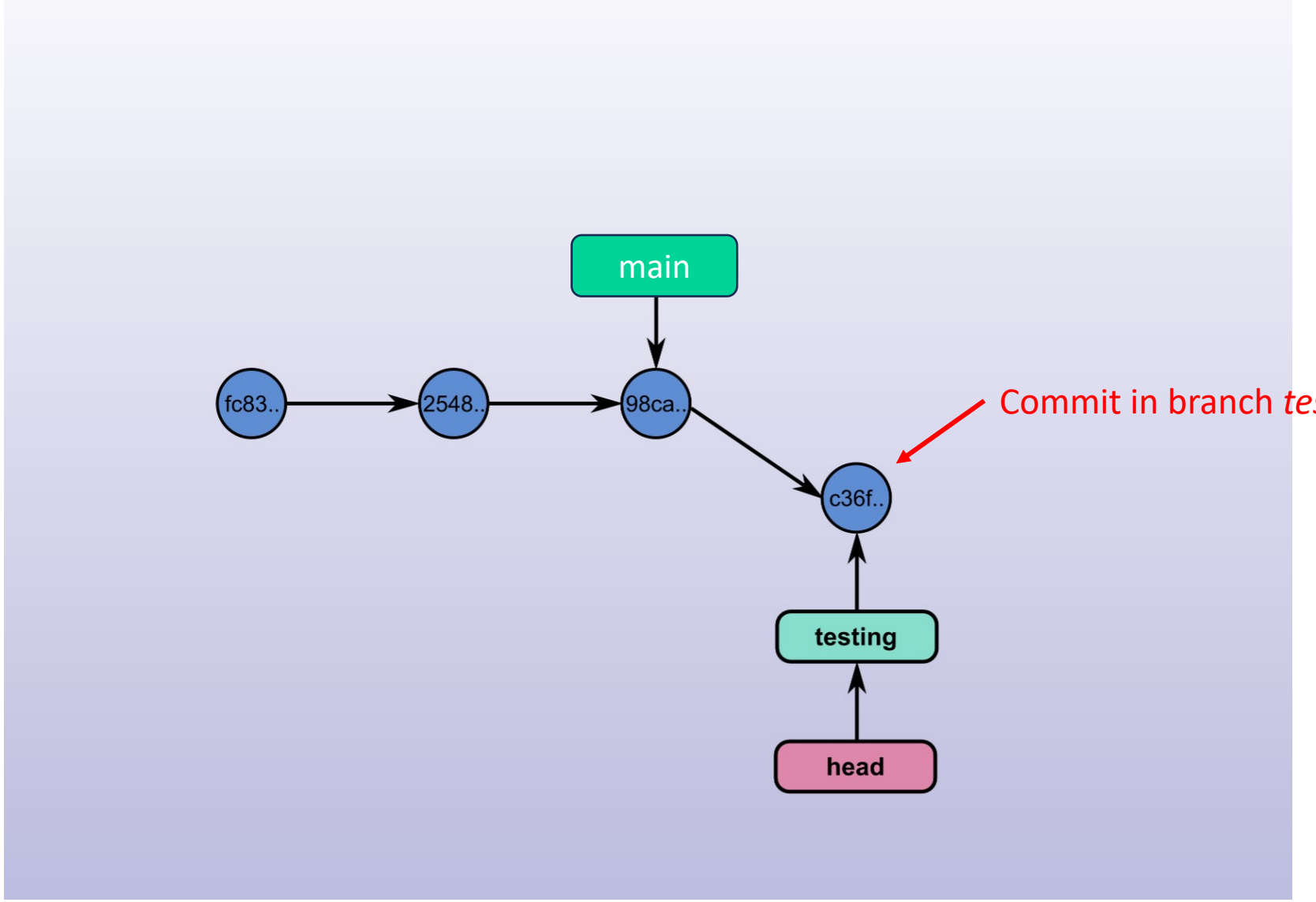
To switch branches

git checkout testing

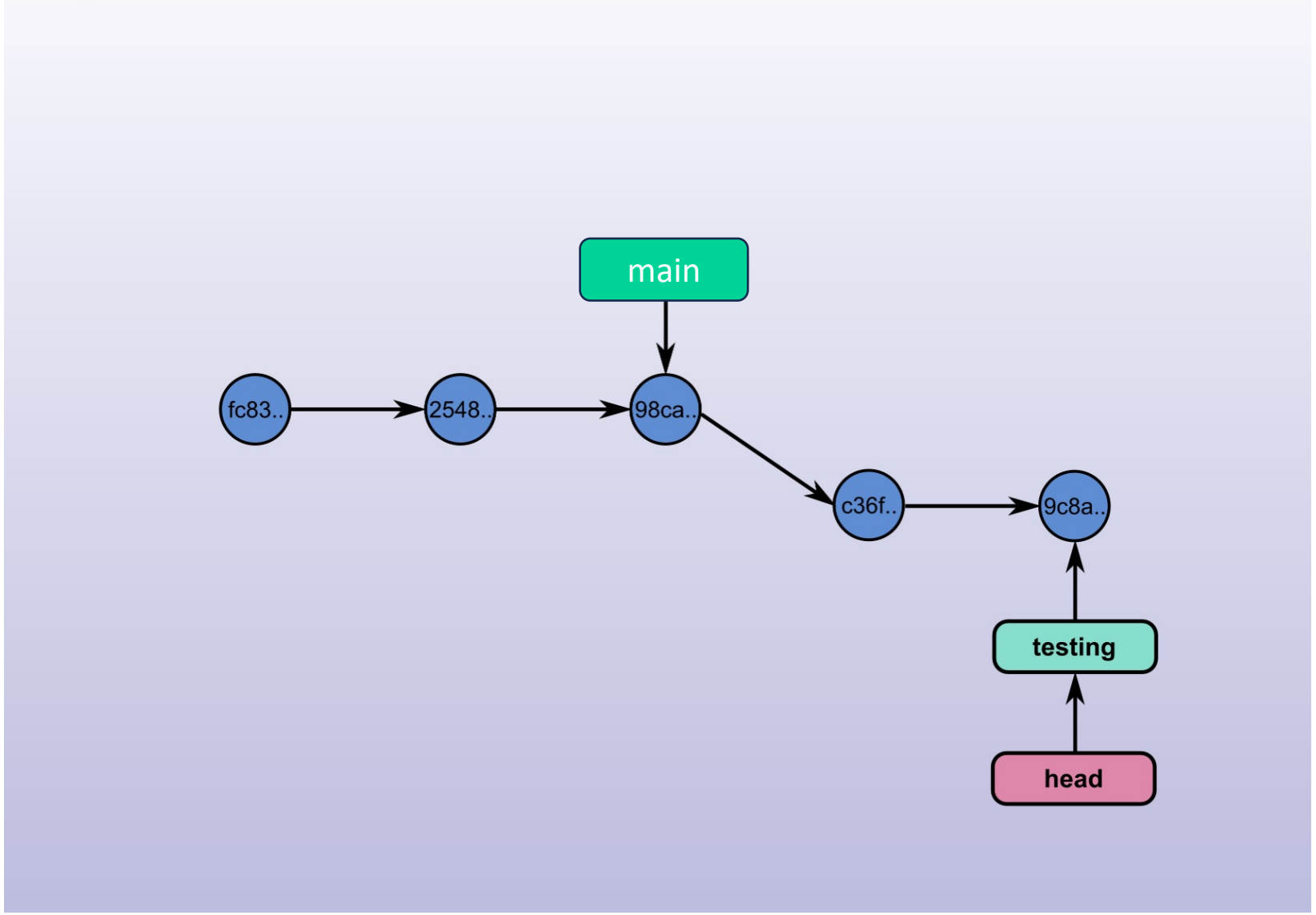
or

git switch testing

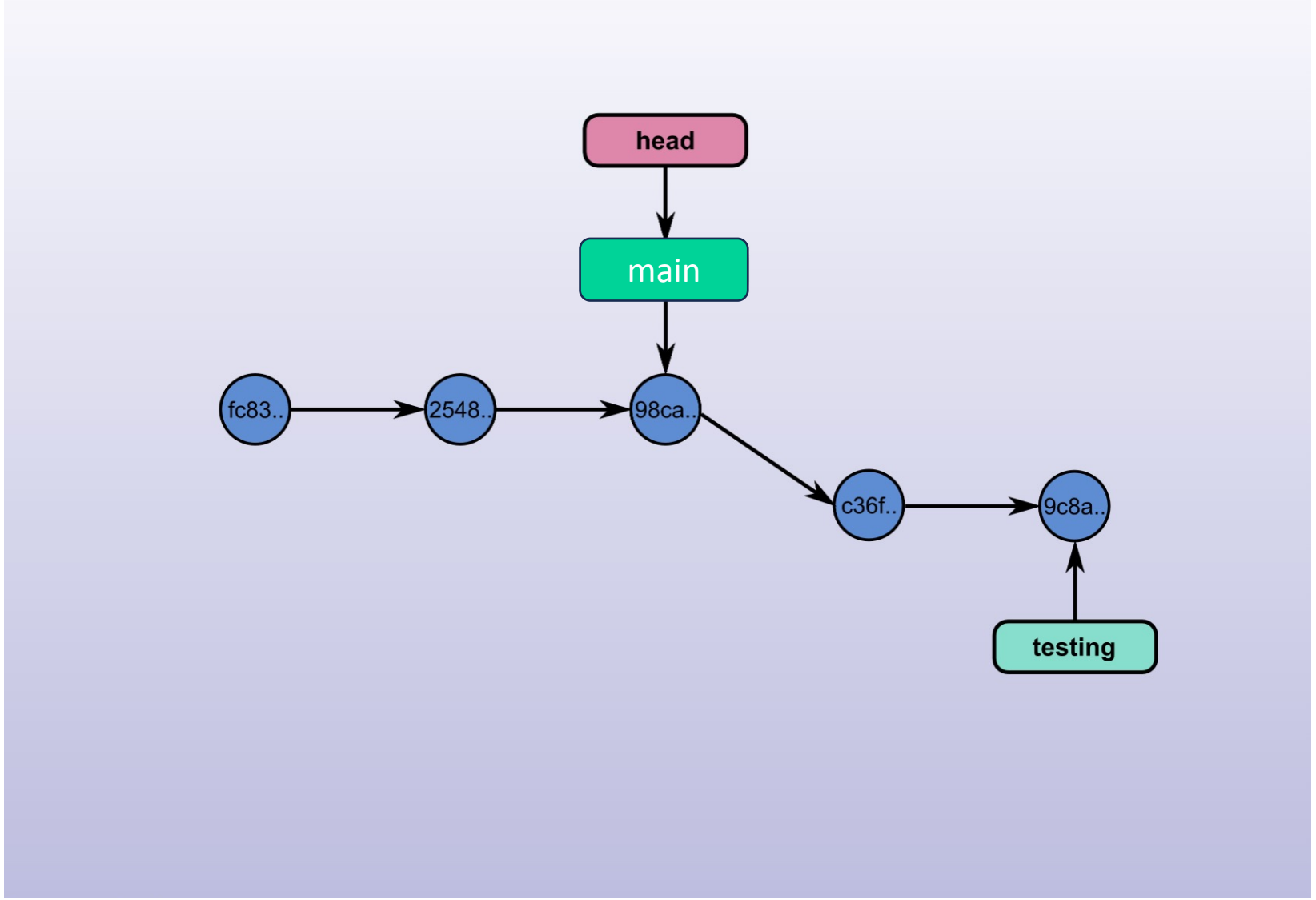
We can switch branches



We can commit in a branch



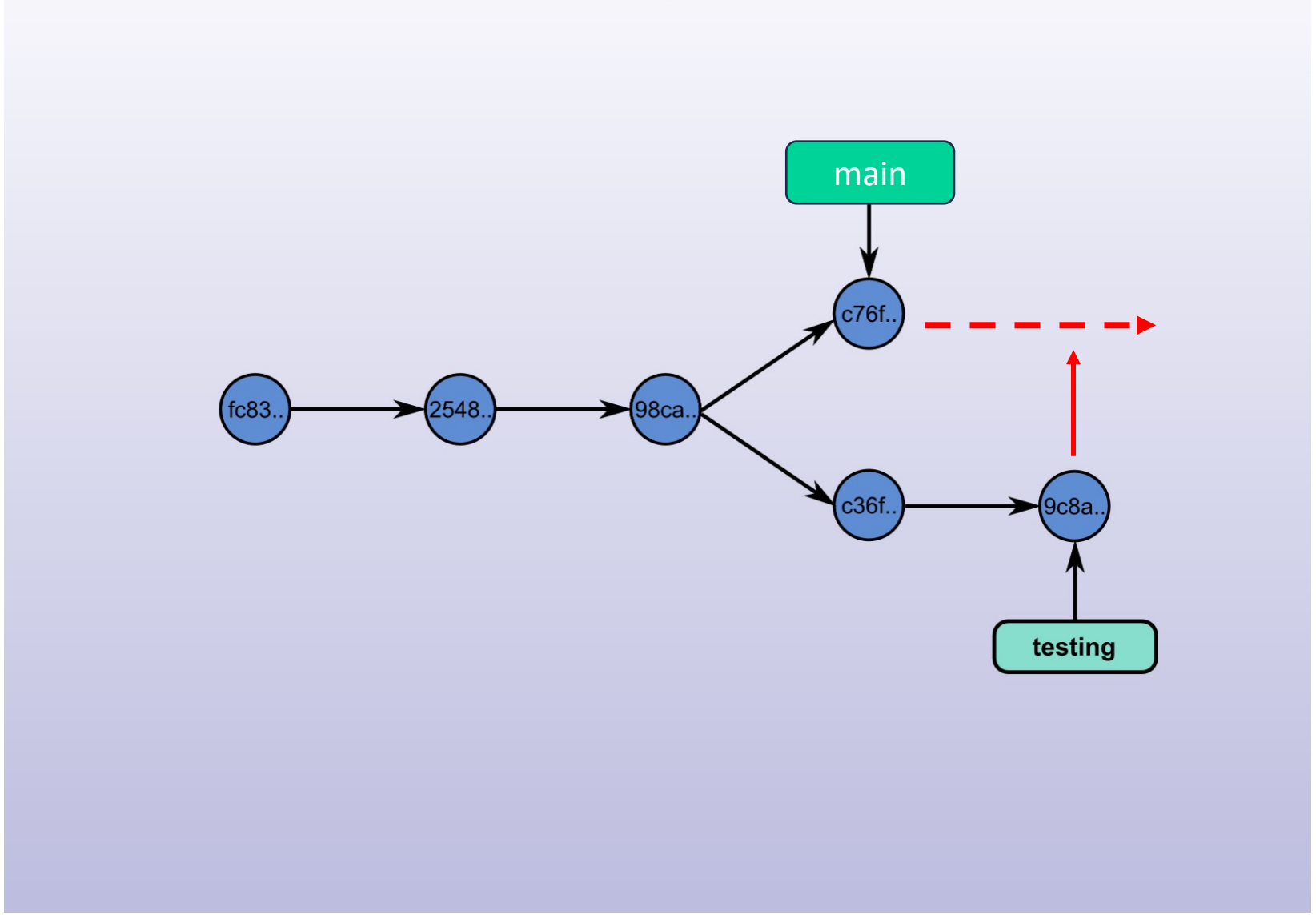
And commit again…



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

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"

Working on a branch. This could be work being done by another developer.

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

Back in main work goes on…

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