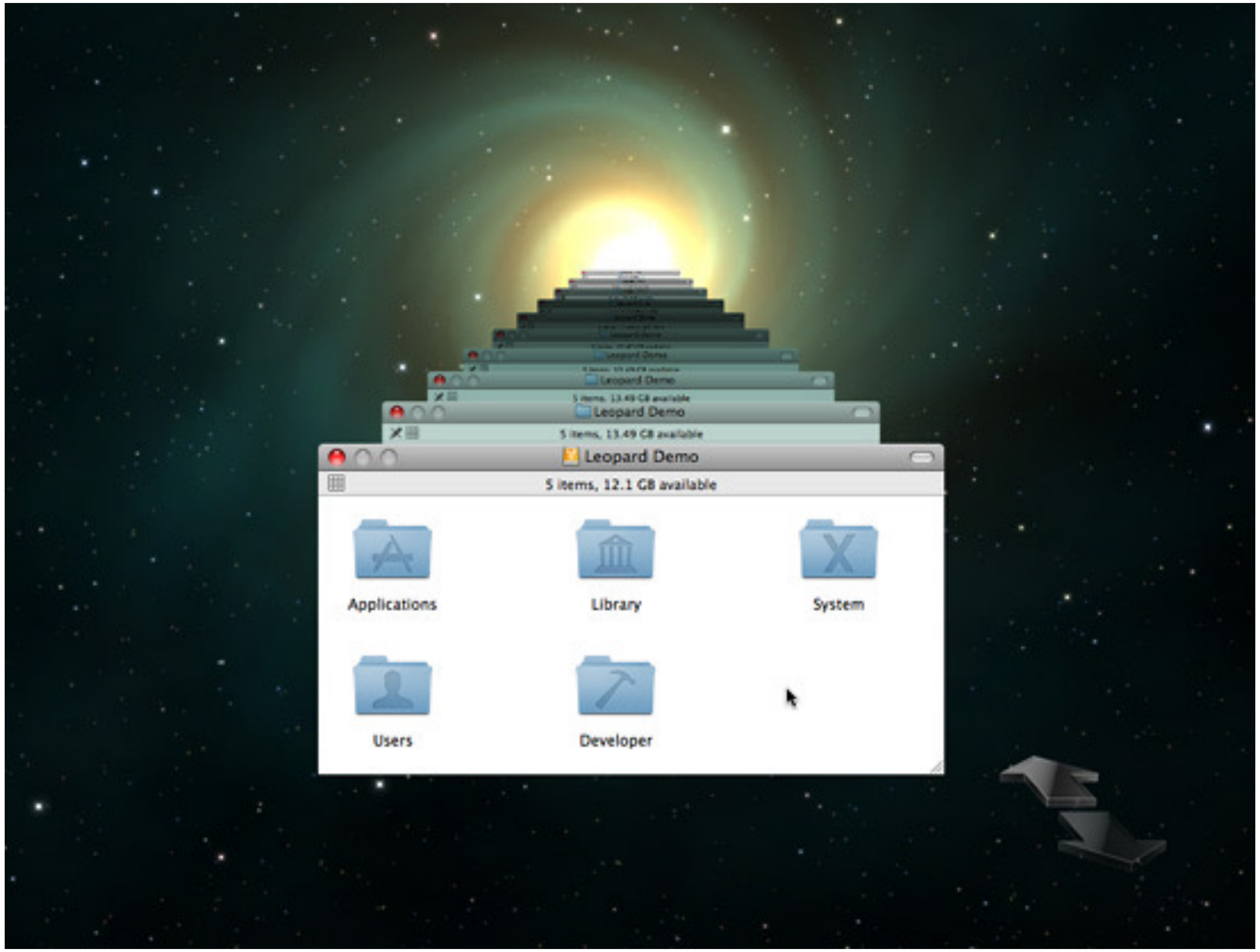
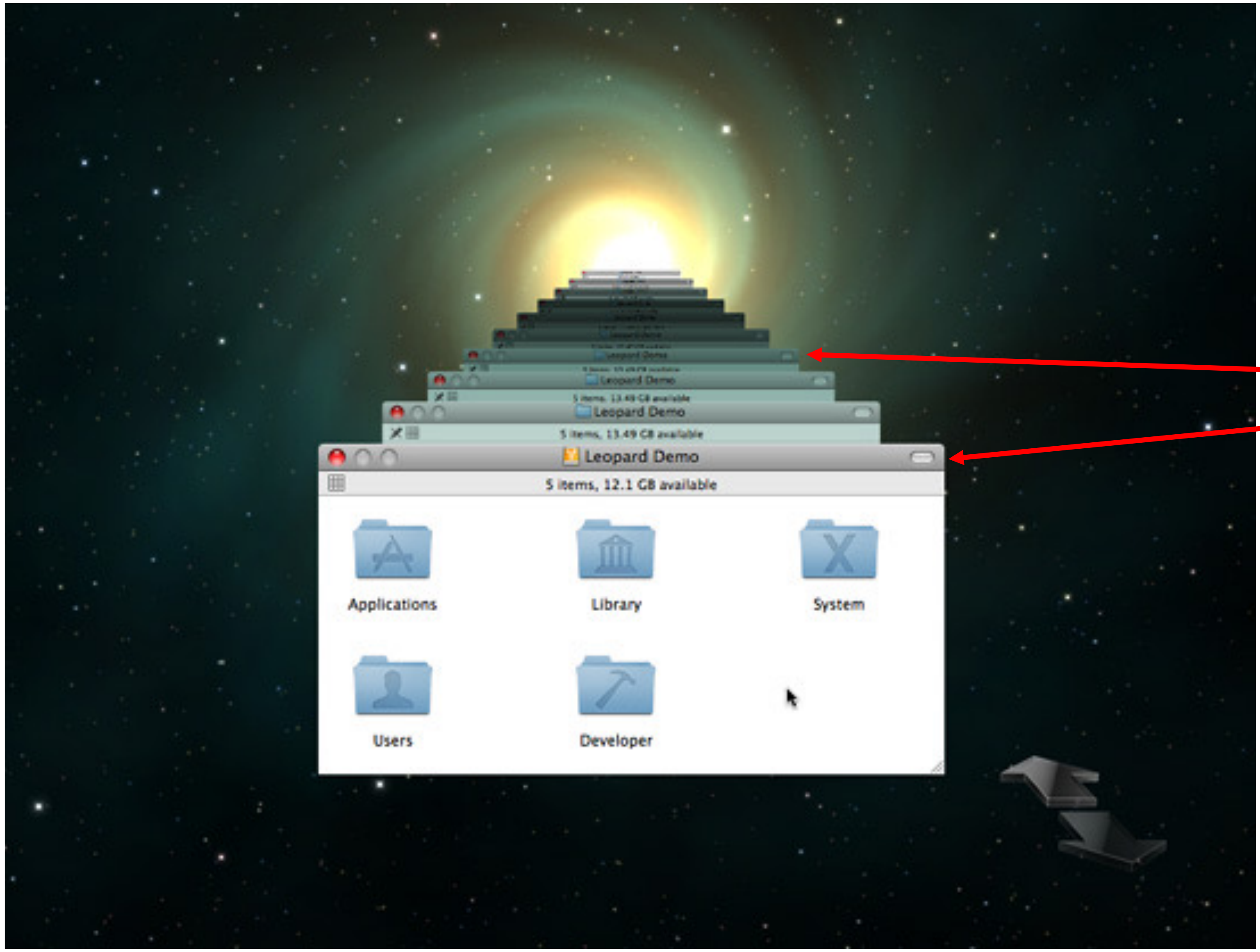
Introduction to Git

Version Control System (VCS)



Version Control System (VCS)



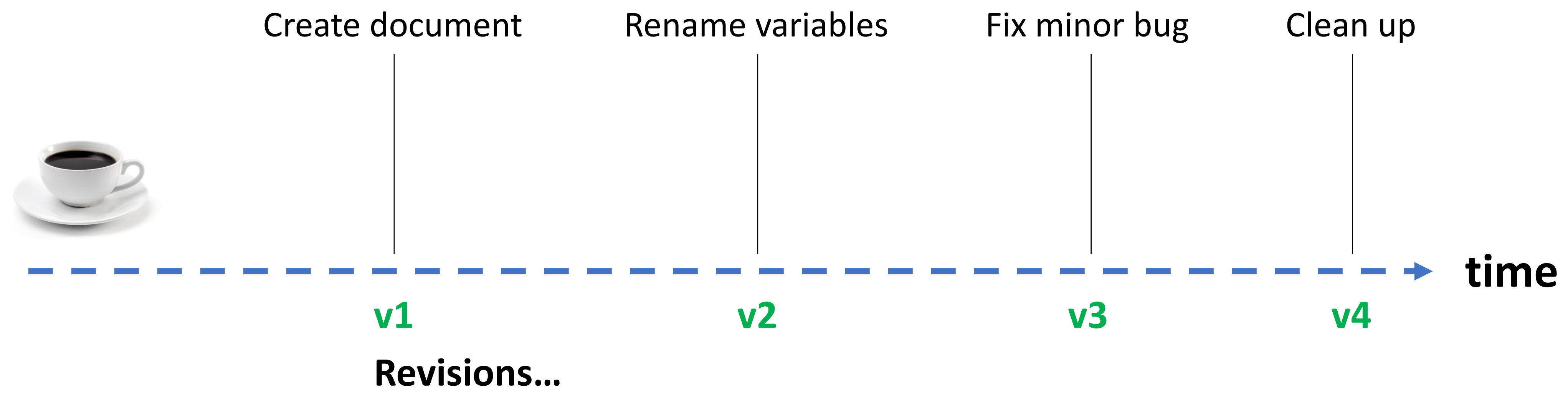
Mac "Time Machine"

Older version

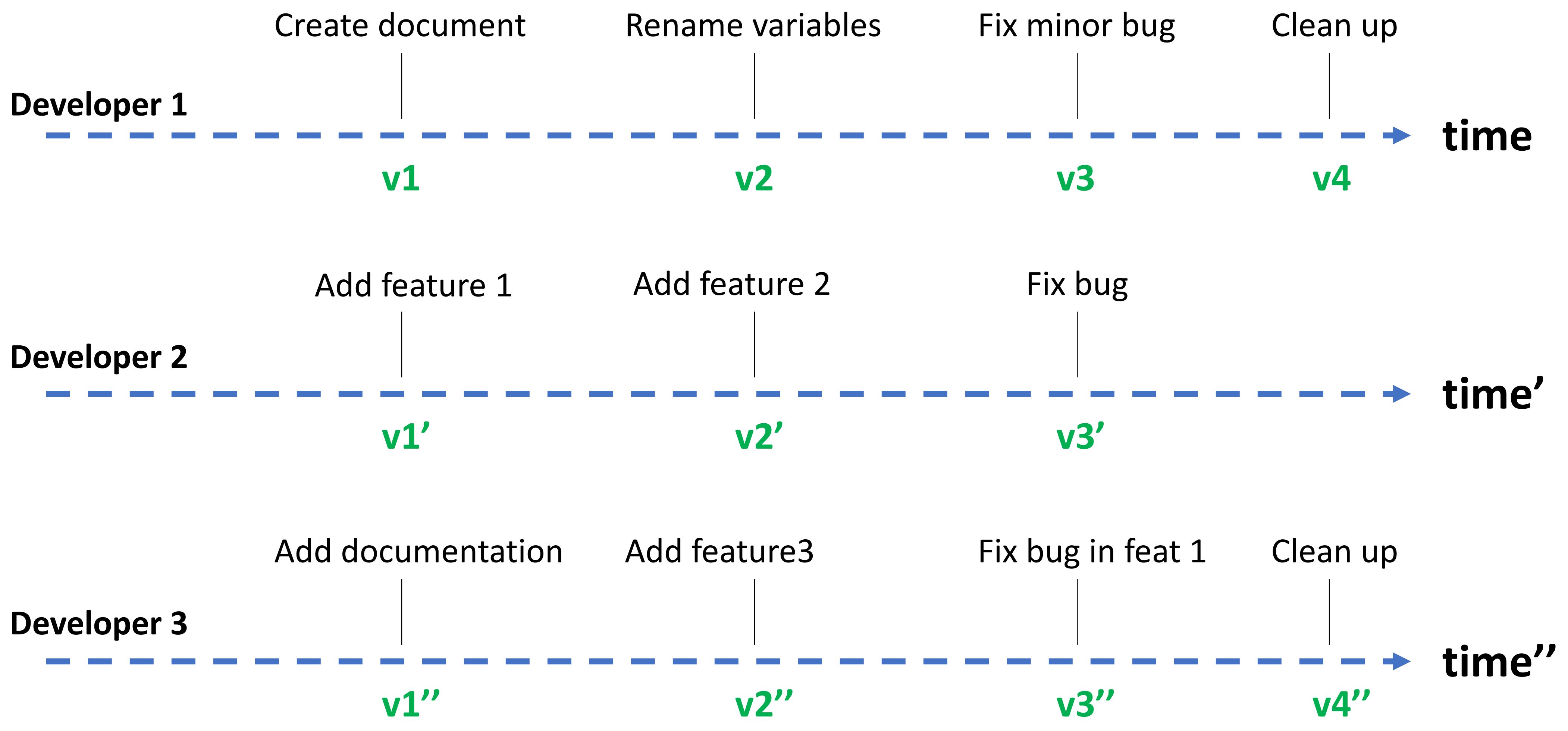
Current version

A ~~Day~~ in the Life of a Software Developer

Week



A File in the Life of a Software Project

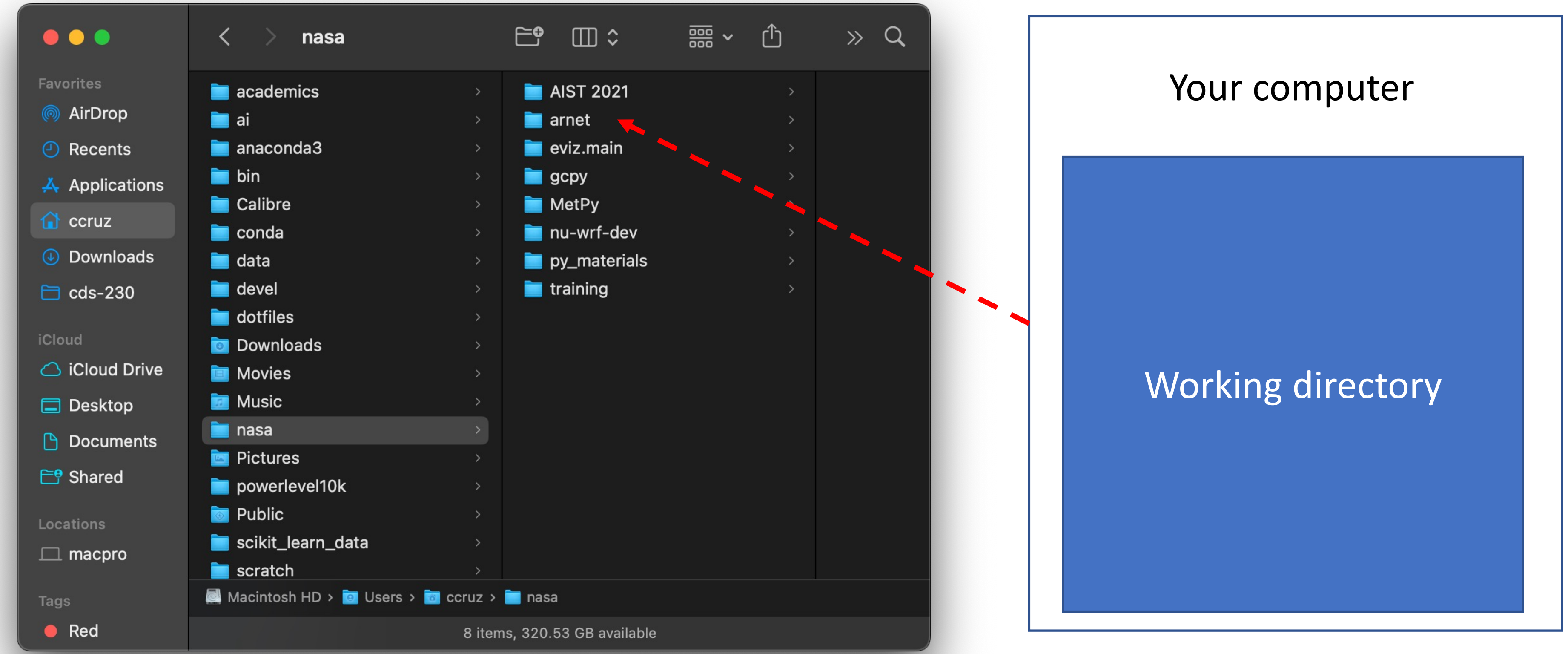


Etc….

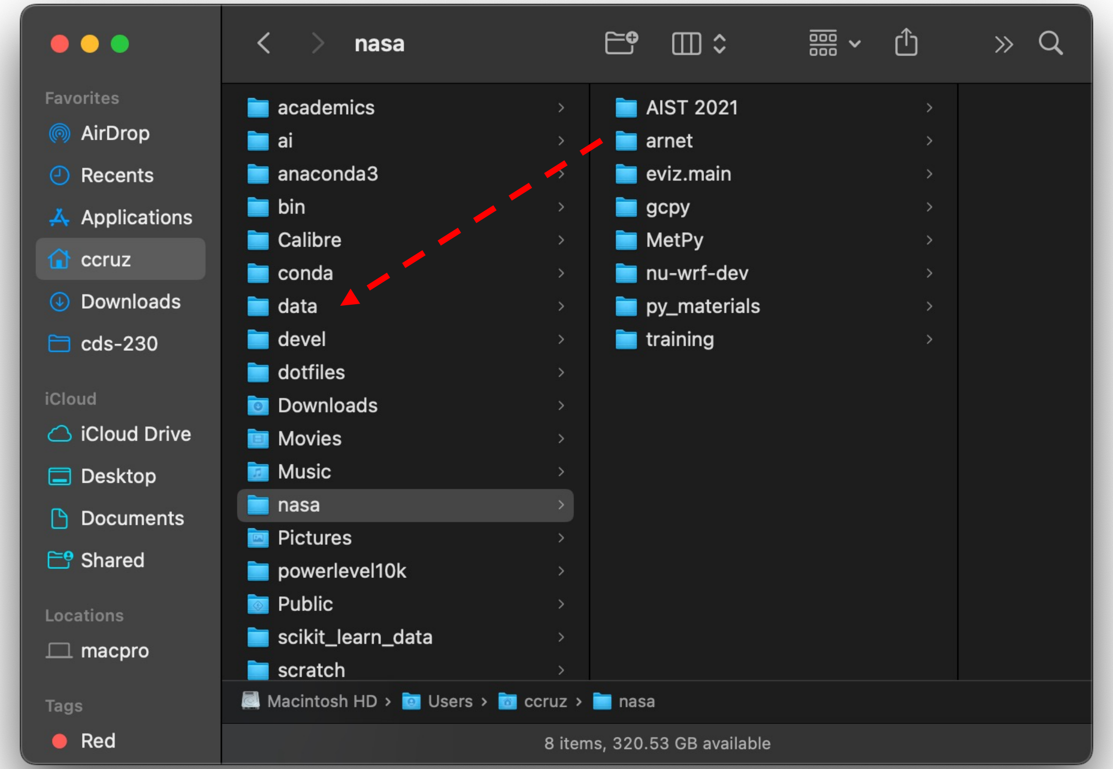
We need to manage code using a VCS

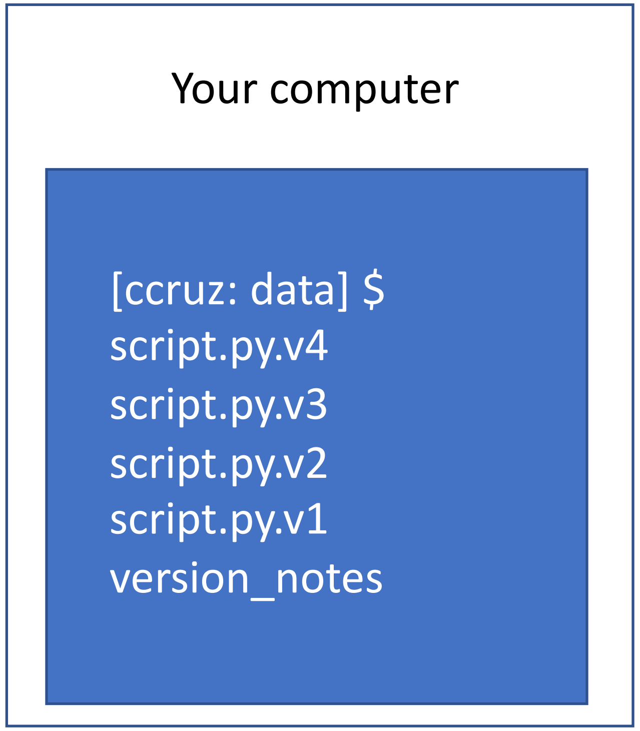
AKA Software Code Management (SCM)

How do we manage source code changes?



How do we manage source code changes?

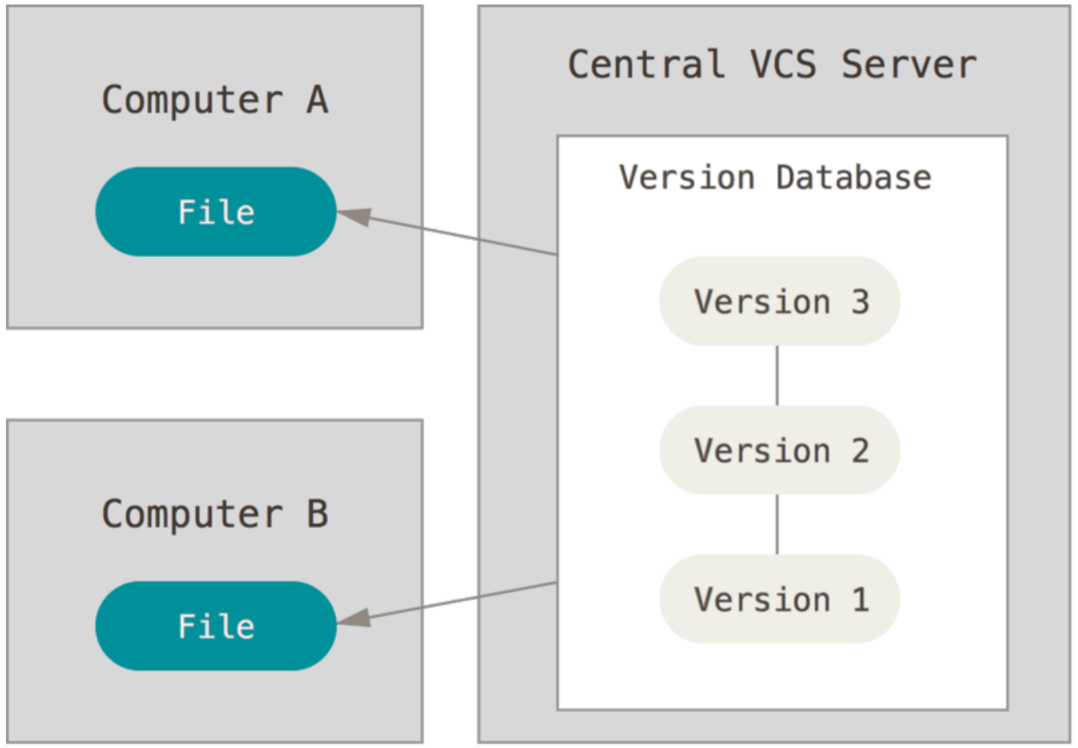




```
[ccruz: data] $
script.py.v4
script.py.v3
script.py.v2
script.py.v1
version_notes
```

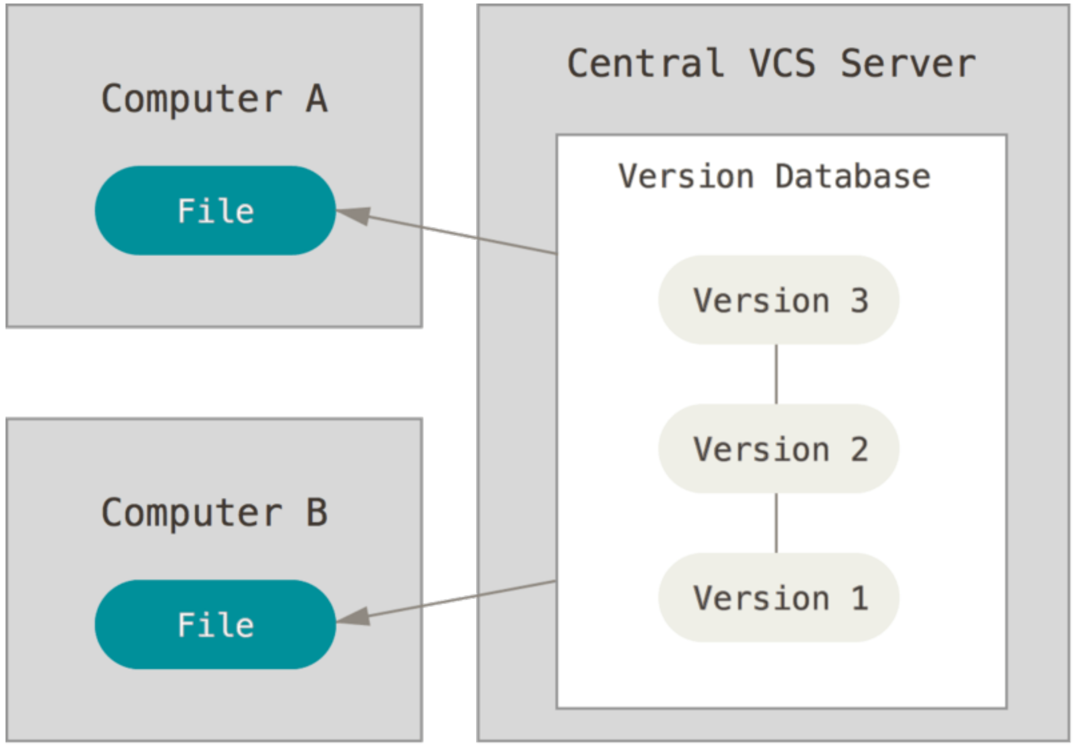
Local "VCS"

How do we manage source code changes? Use a VCS



Centralized VCS

Centralized VCS

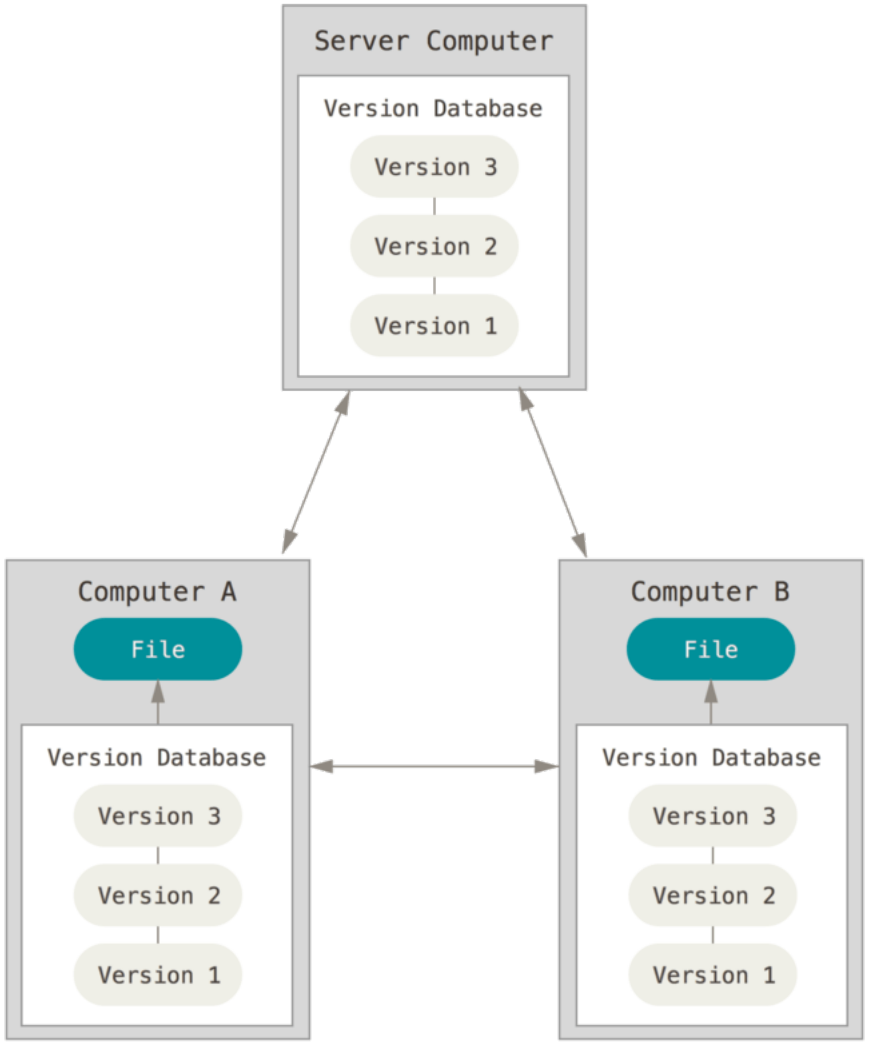


Examples:

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

Distributed VCS

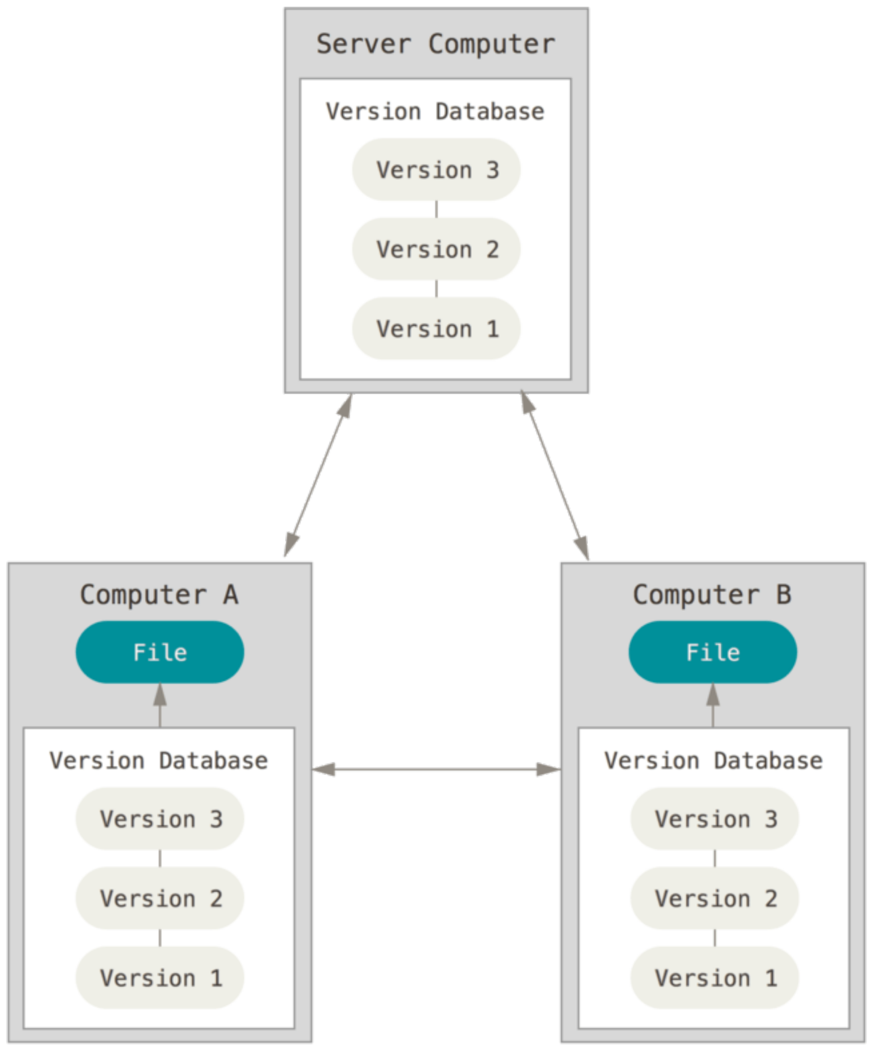
Clone



Clone

Clone

Distributed VCS



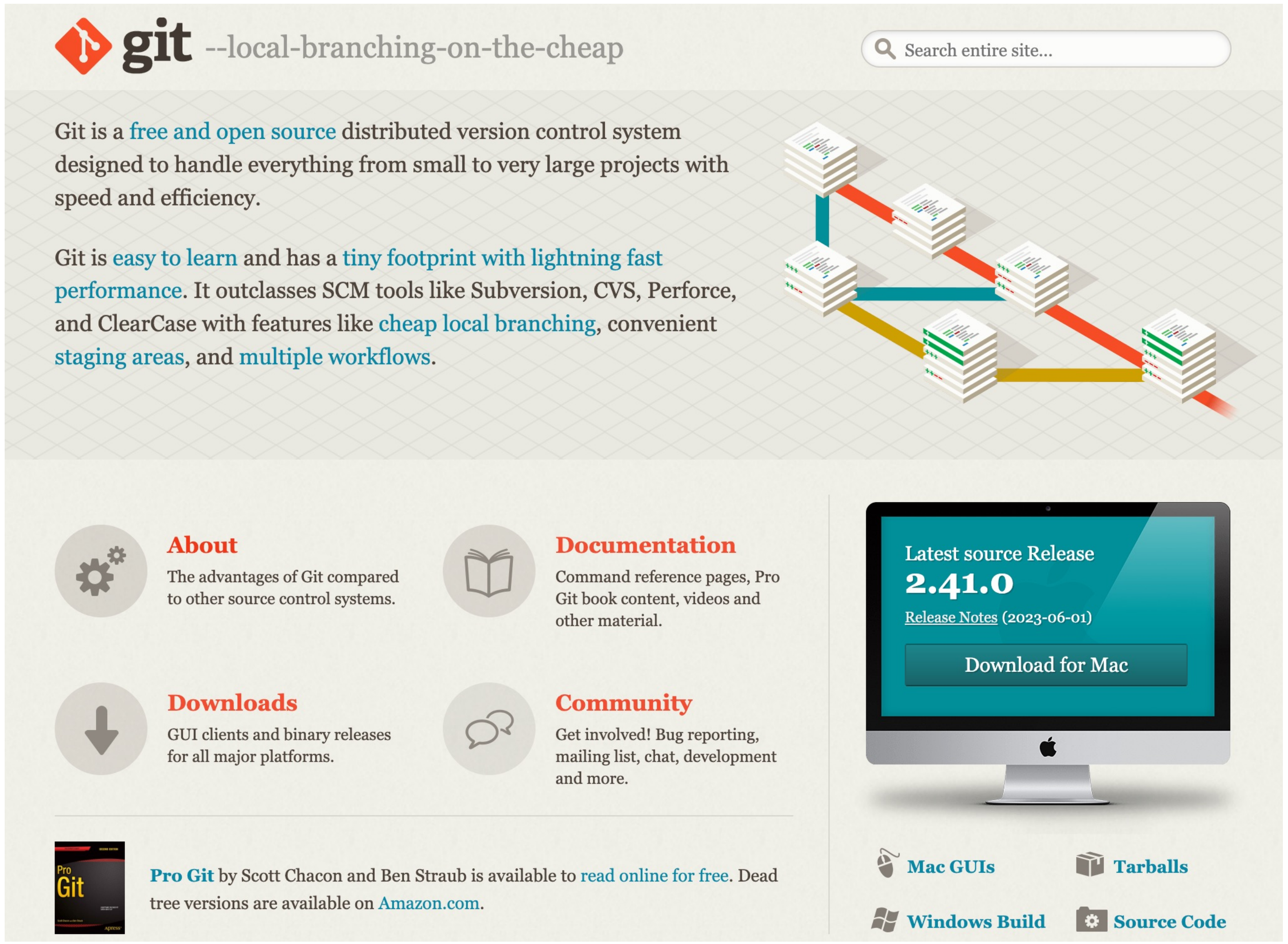
Examples:

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

Benefits of SCM using a VCS

- Integrity and stability of code
- Enables collaboration
- Facilitates project management
- And, of course, version control

What is Git?



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

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

Mac GUIs
Tarballs
Windows Build
Source Code

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

Using Git

1. Install it
2. Interacting with Git
   - Command line
   - GUI
3. Configure Git
   - Create .gitconfig
4. Create a new “repo”
5. Tracking files
   - Create .gitignore
6. Branches

Using Git

1. Install it √
2. Interacting with Git
   - Command line
   - GUI
3. Configure Git
   - Create .gitconfig
4. Create a new “repo”
5. Tracking files
   - Create .gitignore
6. Branches

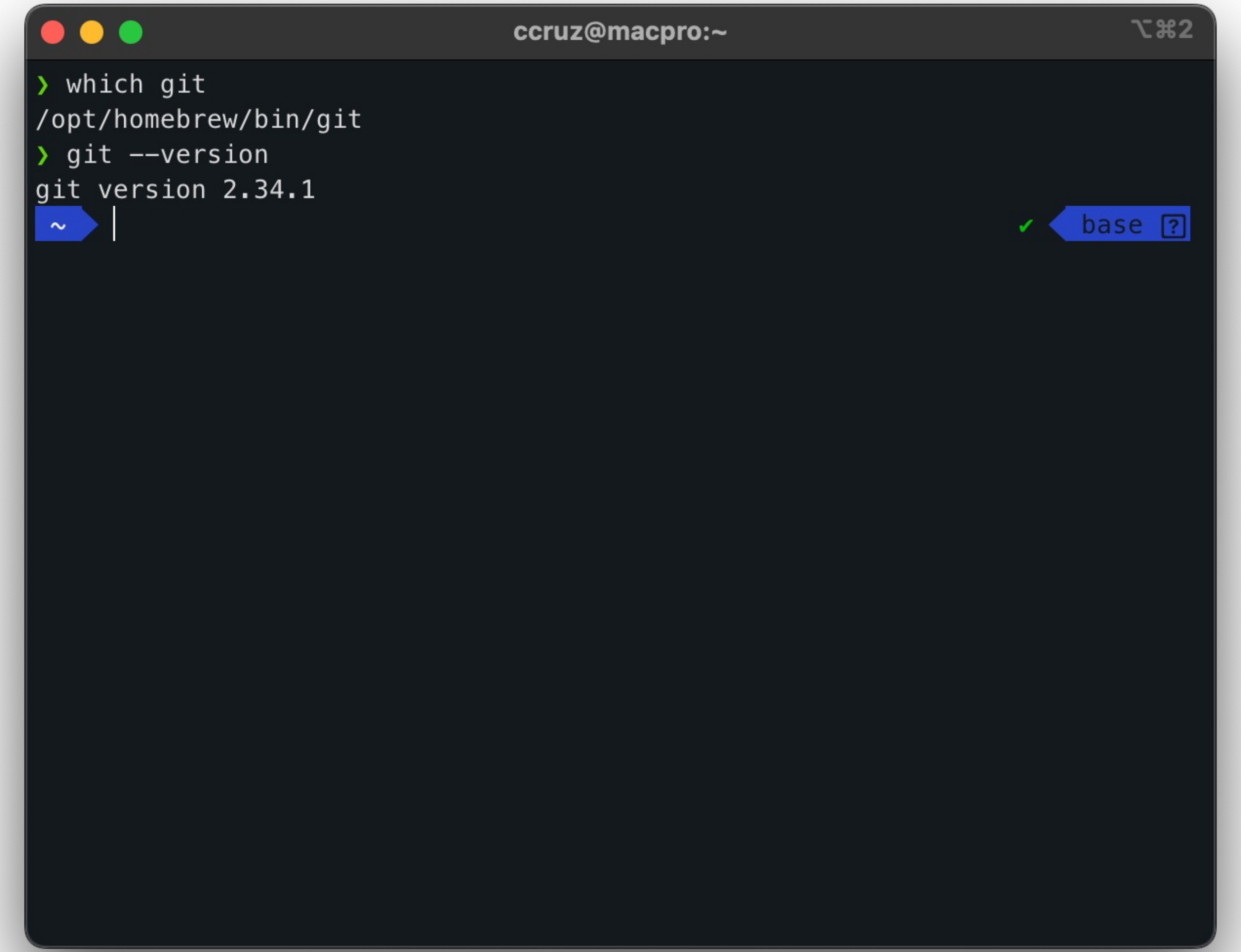
Using Git

1. Install it ✓
2. Interacting with Git
   - Command line
   - ~~GUI~~
3. Configure Git
   - Create .gitconfig
4. Create a new “repo”
5. Tracking files
   - Create .gitignore
6. Branches

Interacting with the Operating System

Shell: software layer to interact with the OS. Examples: bash, zsh, csh

CLI: Command Line Interface



```
❯ which git
/opt/homebrew/bin/git
❯ git --version
git version 2.34.1
~                                                        ✓  base
```

Terminal: Interface for Text commands

Interacting with the Operating System

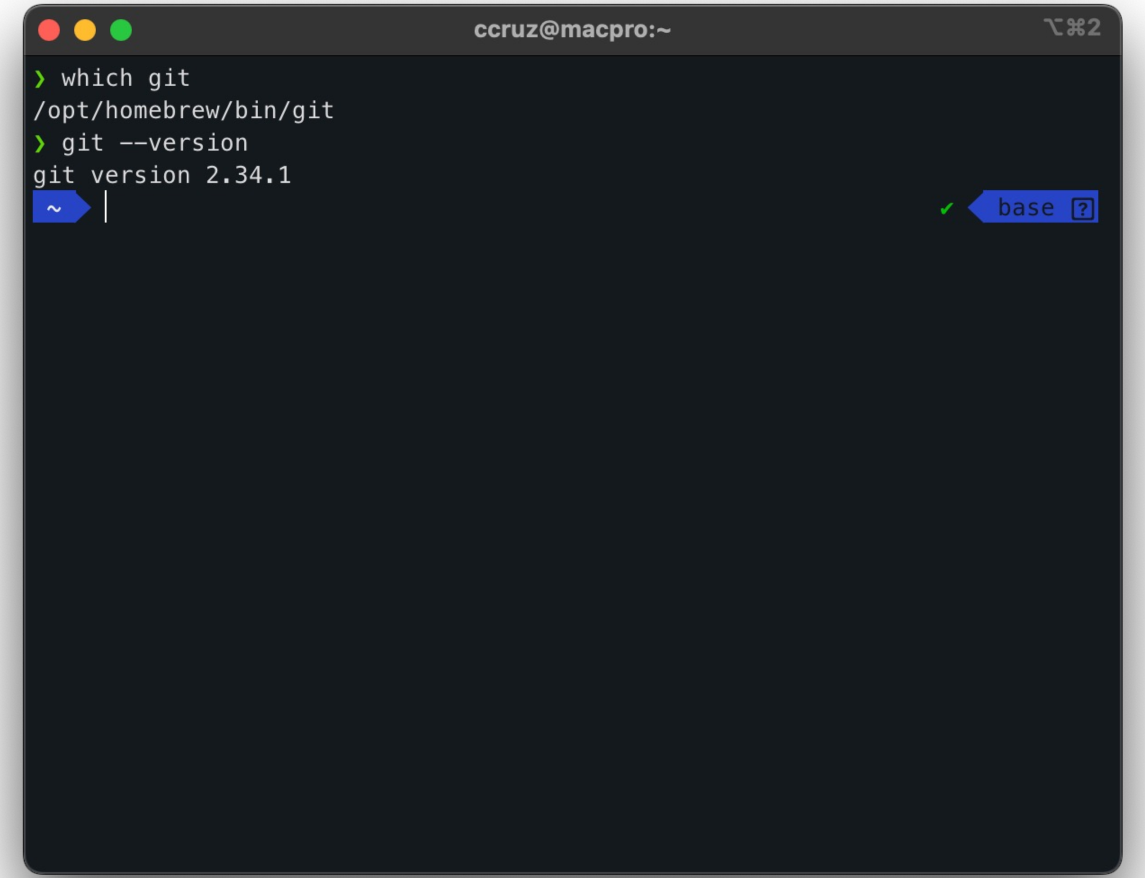
Basic commands:

Directories

pwd : tells you where you currently are (the path)
mkdir dirname : create a new directory
cd dirname : change directory

Files

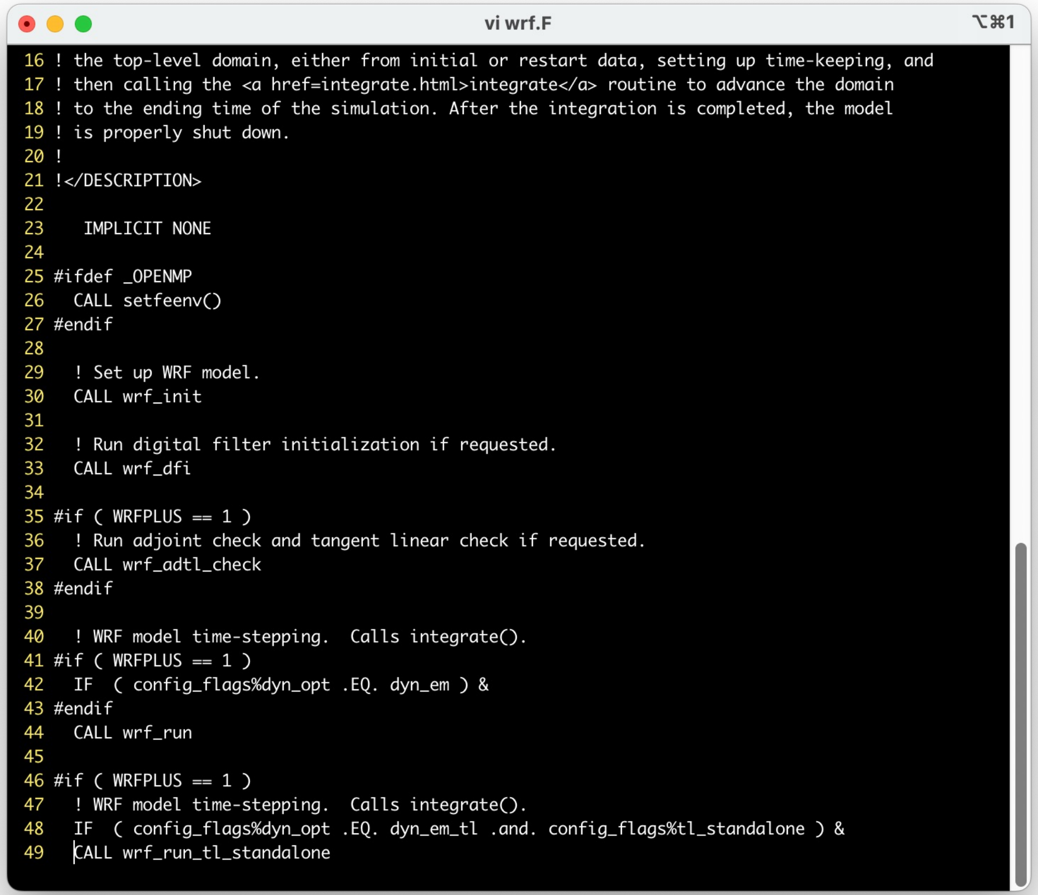
ls : list files
rm filename : remove a file
mv filename1 filename2 : rename a file
diff filename1 filename2 : compare two files
cat filename(s) : print file(s) contents
which : shows path of a command
echo : write to standard output



```
ccruz@macpro:~
❯ which git
/opt/homebrew/bin/git
❯ git --version
git version 2.34.1
~                                                        ✓  base
```

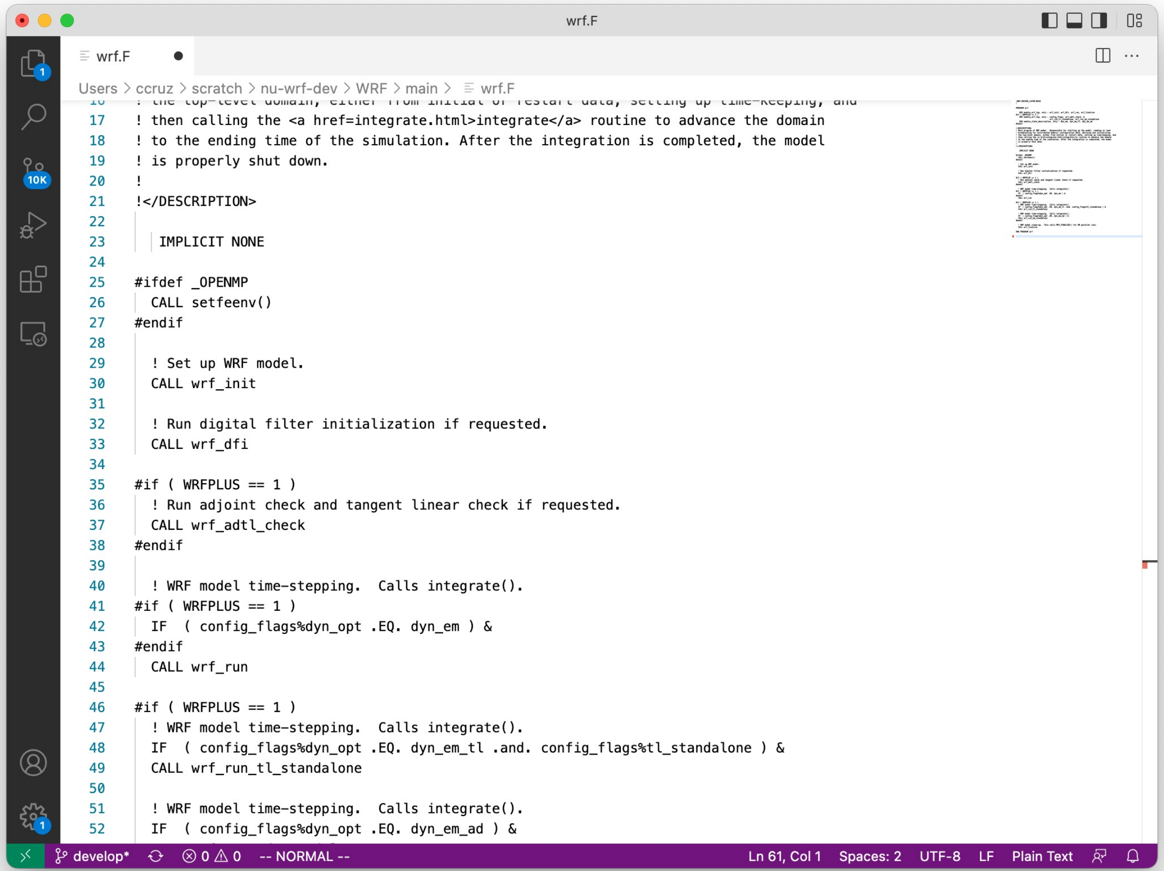
Creating and modifying source code

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


vi wrf.F ⌥⌘1

```
! the top-level domain, either 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

#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
```

vim


wrf.F

wrf.F

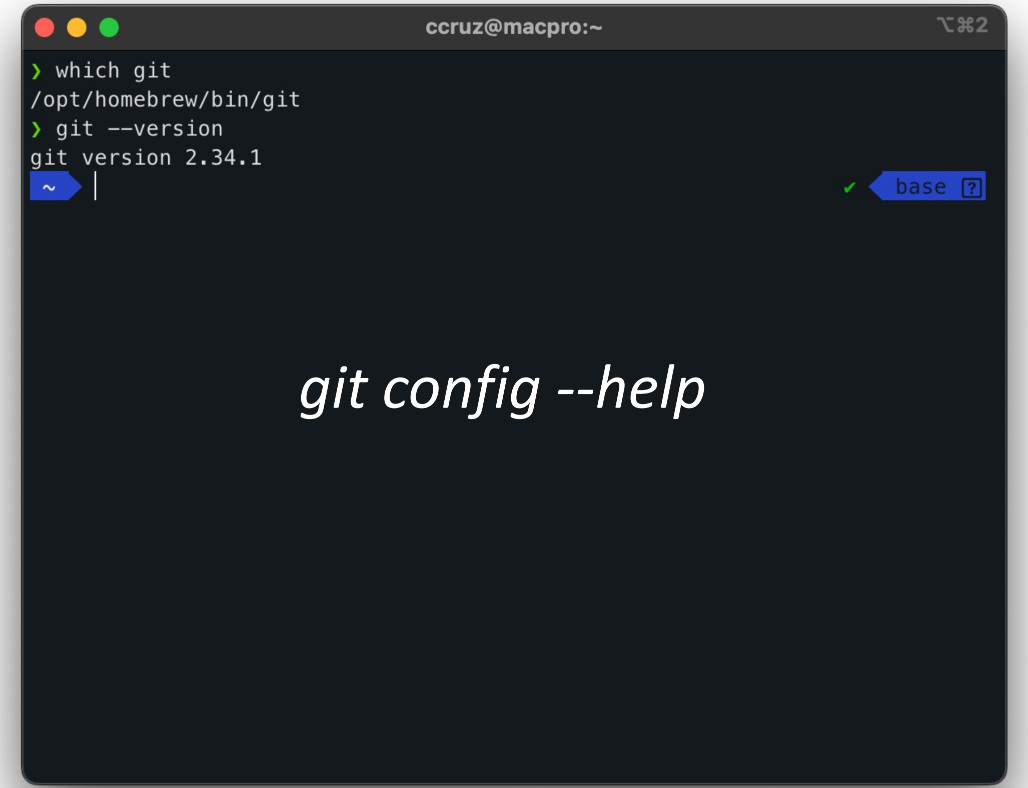
Users > ccruz > scratch > nu-wrf-dev > WRF > main > wrf.F

```
! 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

#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* ⊗ 0 ⚠ 0 -- NORMAL -- Ln 61, Col 1 Spaces: 2 UTF-8 LF Plain Text

Visual Studio

Configuring Git


```
❯ which git
/opt/homebrew/bin/git
❯ git --version
git version 2.34.1
```

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 ”YourFirstName YourLastName"

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 --list

Print config settings

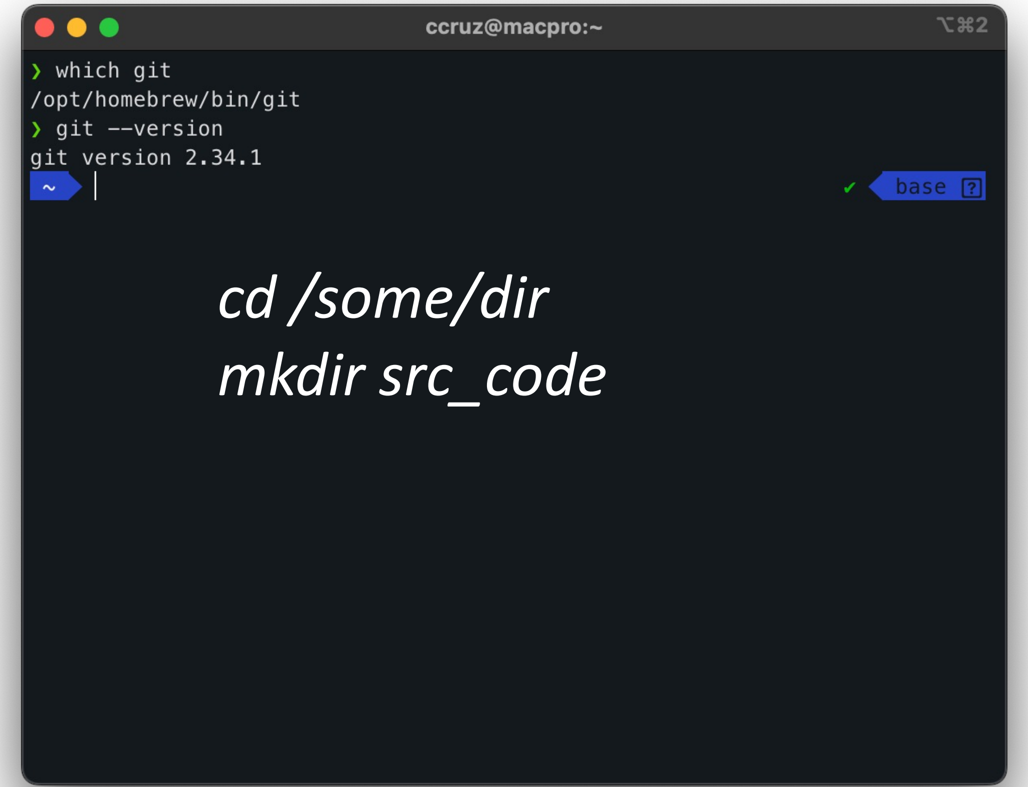
This will create a file named $HOME/.gitconfig with the following contents:

```
[user]
        name = YourFirstName YourLastName
        email = yourusername@domain.com
```

Create a Working Directory*

Exercise

run the following commands on your terminal:


```
ccruz@macpro:~                                        ⌥⌘2
❯ which git
/opt/homebrew/bin/git
❯ git --version
git version 2.34.1
 ~ ❯ |                                        ✔ base ?
```

cd /some/dir

mkdir src_code

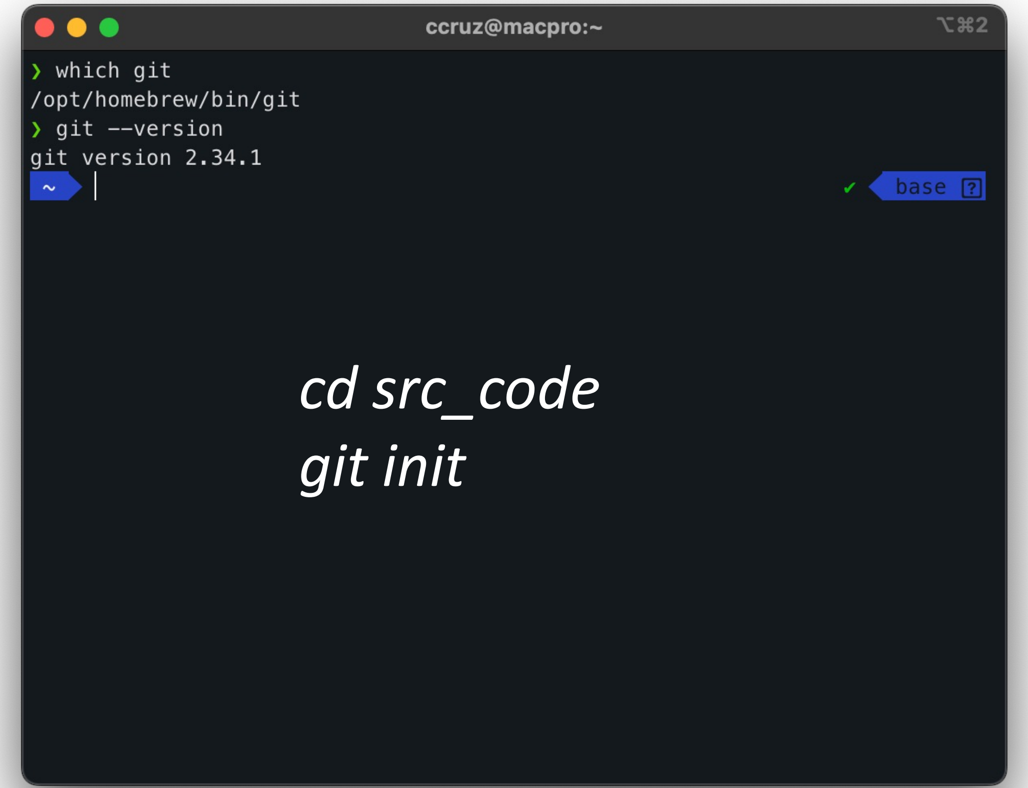
First, go to some directory on your computer - in my case, I use a scratch directory - and create a working directory that we will call ***src_code.***

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

Create an Empty Repository

Exercise

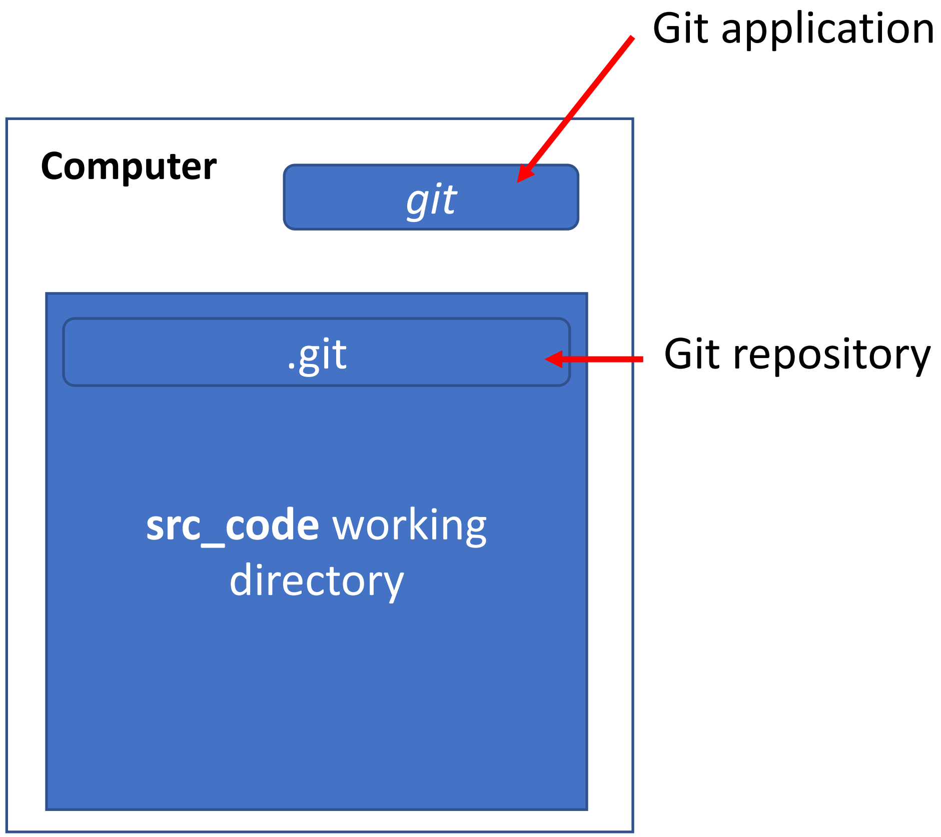
run the following commands on your terminal:



```
❯ which git
/opt/homebrew/bin/git
❯ git --version
git version 2.34.1
~ |                                              ✔ base
```

cd src_code

git init



init: creates a Git repository called *.git*

Demo

Create a file

Exercise

- Open a **Terminal** and change to the ***src_code*** directory you just created
- Create a file named hello.py as follows:

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

- Verify its contents by running

```
cat hello.py
```

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***.

Git
Local repository
Working directory

init
status
add
commit
log
diff
reset

Create repositories

$ git init [project-name] ✓

Creates a new local repository with the specified name

Make changes

$ git status

Lists all new or modified files to be committed

$ git add [file]

Snapshots the file in preparation for versioning

$ git commit -m "[descriptive message]"

Records file snapshots permanently in version history

$ git log

List version history

$ git diff

Shows file differences not yet staged

$ git reset [file]

Unstages the file, but preserve its contents

Editing Files

Exercise

Edit hello.py so that it reads:

```
print('Hello, world!’)
```

- Save. Now run

```
git diff hello.py
```

Add and commit file

Let’s check the history. Run

```
git log
git slog
git hist
```

Git Aliases

Exercise

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

```
git config --global alias.co "checkout"
git config --global alias.ci "commit"
git config --global alias.st "status"
git config --global alias.cm "commit -m"
git config --global alias.cam "commit -am"
git config --global alias.slog "log --oneline --topo-order --graph"
git config --global alias.hist 'log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short'
```

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

```
[alias]
        st = status
        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:
  - *.pyc, .o, .log* files
  - *.DS_Store* hidden files
  - Python-generated directories _ _*pycache*_ _
  - 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 the src_code directory. Its contents should be:

```
*.pyc
_ _pycache_ _
*.log
```

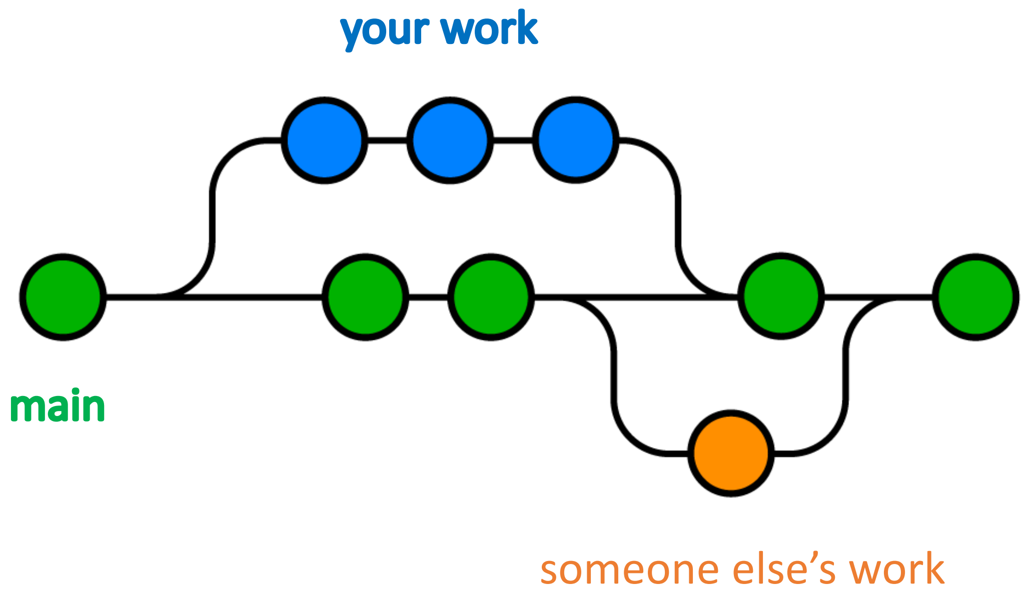
Add and commit to your repo.

More Git Operations

- Adding more files
- Creating directories
- Moving files (git mv)
- Saving your changes (git stash)
- Going back in time (git log + git checkout)
- Undoing changes (git checkout, git clean, git rm)

Branches

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



Basic 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

References

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

Linus Torvalds on git:
https://www.youtube.com/watch?v=4XpnKHJAok8&t=100s&ab_channel=Google

Basic terminal commands:
https://ubuntu.com/tutorials/command-line-for-beginners#1-overview