Scientific Computing @ CLASSE

Practical Orientation Sessions

2024-06-07
2024-06-12
2024-06-14

https://wiki.classe.cornell.edu/Computing/SummerStudentOrientation

Overview

- These sessions are a *basic* and *practical* introduction to computing at CLASSE
- Hands-on exercises will help you set up a hospitable environment for working on your project's software (bring your laptop!)
- 2024-06-07: The CLASSE Linux system
    - Logging in, where to work, practice using the terminal
- 2024-06-12: Developing python code
    - Environment management, jupyter, tips on how to write "good" python code
- 2024-06-14: Version control (git)
    - What is version control, how to use git

Scientific Computing @ CLASSE

The CLASSE Linux System

2024-06-07

Overview

- CLASSE Linux computers
- How to log in
- Where to work on your projects
- Practice using the terminal

The CLASSE Linux System

Where to work — computers

- `lnx201.classe.cornell.edu`
  - General purpose login node — a *shared* resource
  - Use for file browsing, text editing, running code that doesn't consume many resources
- CLASSE compute farm
  - https://wiki.classe.cornell.edu/Computing/ComputeFarmIntro
  - Use for running code that may consume a lot of resources
- Station computers: `idxx.classe.cornell.edu`
  - Connected to station hardware (beamstops, motors, detectors, etc.)
  - Use ONLY if you are controlling station hardware
- More guidance on where to work: https://wiki.classe.cornell.edu/Computing/WhichComputer

The CLASSE Linux System

Logging in

`ssh` — **s**ecure **sh**ell

- On Linux & mac terminals:

```
ssh <CLASSEID>@<host>
```

  - For example:

```
ssh kls286@lnx201.classe.cornell.edu
```

On Windows: PuTTY

- https://wiki.classe.cornell.edu/Computing/WinTunnelVncSSH

NoMachine — full desktop interface

- On any computer, open an web browser and visit:

  https://nomachine.classe.cornell.edu

- Be prepared for issues with speed, copy/paste, and dropped connections

Another CLASSE-specific option: https://jupyter01.classe.cornell.edu

The CLASSE Linux System

Exercise 1

Open a terminal on `lnx201` using one of the following options:

1. `ssh` from your computer's terminal (for Mac and Linux users)
2. PuTTY (for Windows users)
3. https://nomachine.classe.cornell.edu
4. https://jupyter01.classe.cornell.edu

The CLASSE Linux System

Where to work — directories

- HomeDisk (`/home/<CLASSE-ID>`) is limited to 1GB
  - This is the landing point when you open a new terminal or log in with ssh
  - ***Do not make a habit of working in your home directory!***
- Recommended working space: `/nfs/`…
  - For CHESS students:`/nfs/chess/user/<CLASSE-ID>/`
  - For other students: ask your project mentor
    - For these exercises, use `/cdat/tem/<CLASSE-ID>/`
  - `nfs` stands for Network File System — if you put something in `/nfs` on `lnx201`, it will also be there on the CLASSE Compute Farm nodes, the station computers, the computers in the CESR and CHESS operations areas, etc.

The CLASSE Linux System

Exercise 2

Make a symbolic link to your `/nfs/chess/user/<CLASSE-ID>/` directory inside your home directory.

1. In the terminal you opened before, run this command (make sure to substitute appropriate values where something is enclosed in `<>` before running):

```
ln -s /cdat/tem/<your CLASSE ID>/ ~/<your link name>
```

The CLASSE Linux System

Exercise 3

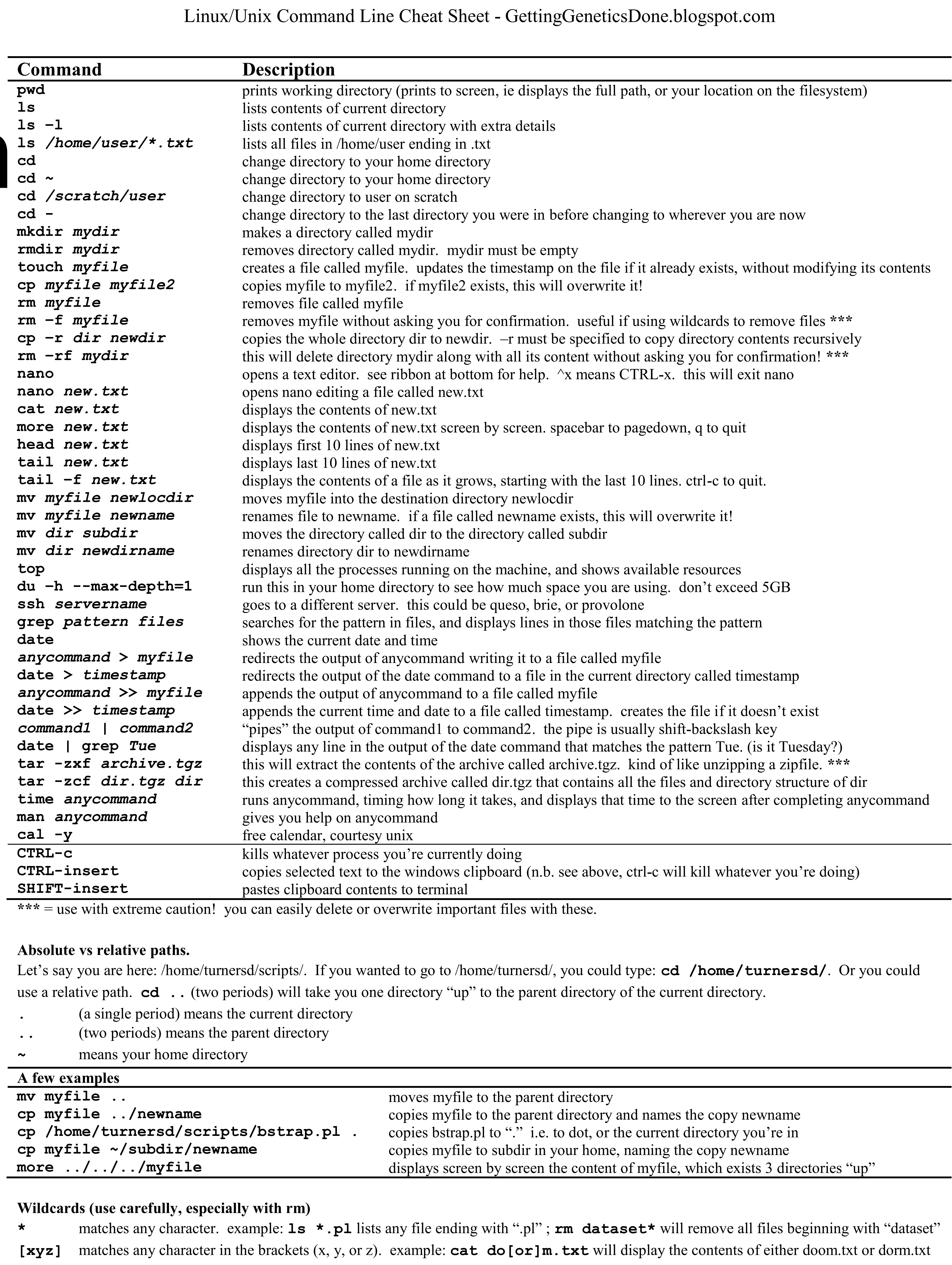
Start navigating in the same terminal as before. Run the following commands:

1. `pwd`
   - “Print working directory” tells you what the current working directory is
2. `ls`
   - Lists the contents of the current working directory
3. `ls -la`
   - Lists the contents of the current working directory with the additional options:
     - `-l` tells `ls` to show details about each file’s type, permissions, size, etc. (“l” for “long”)
     - `-a` tells `ls` to list hidden files, too (“a” for “all”)
   - Individual options to `ls` like `-l` and `-a` can be shortened to `-la`
4. `cd <your link name from Exercise 2>`
   - “change directory” changes your current working directory to the specified destination

The CLASSE Linux System

Getting comfortable in the terminal

- A fantastic intro to Linux in general and at CLASSE: https://xcitecourse.org/theme2/sf100/linux-commandline-scripting
  - Thanks to our collaborators from X-CITE (CyberInfrastructure Training and Education for Synchrotron X-Ray Science)!
- A nice cheat sheet of Linux commands —>
- If you don't know how to use a command, try running one of the following to get a help menu / manual entry for it:
  - `<command> —h`
  - `<command> —help`
  - `man <command>`



Linux/Unix Command Line Cheat Sheet - GettingGeneticsDone.blogspot.com

Command	Description
`pwd`	prints working directory (prints to screen, ie displays the full path, or your location on the filesystem)
`ls`	lists contents of current directory
`ls –l`	lists contents of current directory with extra details
`ls /home/user/*.txt`	lists all files in /home/user ending in .txt
`cd`	change directory to your home directory
`cd ~`	change directory to your home directory
`cd /scratch/user`	change directory to user on scratch
`cd -`	change directory to the last directory you were in before changing to wherever you are now
`mkdir mydir`	makes a directory called mydir
`rmdir mydir`	removes directory called mydir. mydir must be empty
`touch myfile`	creates a file called myfile. updates the timestamp on the file if it already exists, without modifying its contents
`cp myfile myfile2`	copies myfile to myfile2. if myfile2 exists, this will overwrite it!
`rm myfile`	removes file called myfile
`rm –f myfile`	removes myfile without asking you for confirmation. useful if using wildcards to remove files ***
`cp –r dir newdir`	copies the whole directory dir to newdir. –r must be specified to copy directory contents recursively
`rm –rf mydir`	this will delete directory mydir along with all its content without asking you for confirmation! ***
`nano`	opens a text editor. see ribbon at bottom for help. ^x means CTRL-x. this will exit nano
`nano new.txt`	opens nano editing a file called new.txt
`cat new.txt`	displays the contents of new.txt
`more new.txt`	displays the contents of new.txt screen by screen. spacebar to pagedown, q to quit
`head new.txt`	displays first 10 lines of new.txt
`tail new.txt`	displays last 10 lines of new.txt
`tail –f new.txt`	displays the contents of a file as it grows, starting with the last 10 lines. ctrl-c to quit.
`mv myfile newlocdir`	moves myfile into the destination directory newlocdir
`mv myfile newname`	renames file to newname. if a file called newname exists, this will overwrite it!
`mv dir subdir`	moves the directory called dir to the directory called subdir
`mv dir newdirname`	renames directory dir to newdirname
`top`	displays all the processes running on the machine, and shows available resources
`du –h --max-depth=1`	run this in your home directory to see how much space you are using. don't exceed 5GB
`ssh servername`	goes to a different server. this could be queso, brie, or provolone
`grep pattern files`	searches for the pattern in files, and displays lines in those files matching the pattern
`date`	shows the current date and time
`anycommand > myfile`	redirects the output of anycommand writing it to a file called myfile
`date > timestamp`	redirects the output of the date command to a file in the current directory called timestamp
`anycommand >> myfile`	appends the output of anycommand to a file called myfile
`date >> timestamp`	appends the current time and date to a file called timestamp. creates the file if it doesn't exist
`command1 | command2`	"pipes" the output of command1 to command2. the pipe is usually shift-backslash key
`date | grep Tue`	displays any line in the output of the date command that matches the pattern Tue. (is it Tuesday?)
`tar -zxf archive.tgz`	this will extract the contents of the archive called archive.tgz. kind of like unzipping a zipfile. ***
`tar -zcf dir.tgz dir`	this creates a compressed archive called dir.tgz that contains all the files and directory structure of dir
`time anycommand`	runs anycommand, timing how long it takes, and displays that time to the screen after completing anycommand
`man anycommand`	gives you help on anycommand
`cal -y`	free calendar, courtesy unix
`CTRL-c`	kills whatever process you're currently doing
`CTRL-insert`	copies selected text to the windows clipboard (n.b. see above, ctrl-c will kill whatever you're doing)
`SHIFT-insert`	pastes clipboard contents to terminal

*** = use with extreme caution! you can easily delete or overwrite important files with these.

Absolute vs relative paths.

Let's say you are here: /home/turnersd/scripts/. If you wanted to go to /home/turnersd/, you could type: `cd /home/turnersd/`. Or you could use a relative path. `cd ..` (two periods) will take you one directory "up" to the parent directory of the current directory.

`.`	(a single period) means the current directory
`..`	(two periods) means the parent directory
`~`	means your home directory

A few examples	
`mv myfile ..`	moves myfile to the parent directory
`cp myfile ../newname`	copies myfile to the parent directory and names the copy newname
`cp /home/turnersd/scripts/bstrap.pl .`	copies bstrap.pl to "." i.e. to dot, or the current directory you're in
`cp myfile ~/subdir/newname`	copies myfile to subdir in your home, naming the copy newname
`more ../../../myfile`	displays screen by screen the content of myfile, which exists 3 directories "up"

Wildcards (use carefully, especially with rm)

`*`	matches any character. example: `ls *.pl` lists any file ending with ".pl" ; `rm dataset*` will remove all files beginning with "dataset"
`[xyz]`	matches any character in the brackets (x, y, or z). example: `cat do[or]m.txt` will display the contents of either doom.txt or dorm.txt

The CLASSE Linux System

Getting comfortable in the terminal

- Use tab completion so you don’t have to type out long file names or commands

- *DO* make liberal use of your favorite search engine…but *DO NOT* copy / paste commands you find on the internet without understanding what they do and how they work.

- `nano`, `emacs`, and `vi` are good options for editing files in the terminal

- atom and gedit are good options for editing files with a GUI on CLASSE Linux machines

The CLASSE Linux System

Summary

- Log in with `ssh` for terminal access, NoMachine for full graphical desktop
- Use `lnx201` for everyday tasks, the CLASSE Compute Farm for resource-intensive jobs
  - For hardware control & other specialty tasks, ask your project mentor
- Do not put files in your home directory!
  - CHESS students — work in `/nfs/chess/user/<CLASSE-ID>/`
- See https://xcitecourse.org/theme2/sf100/linux-commandline-scripting for more guided materials on how to use the Linux command line at CLASSE

The CLASSE Linux System

Demonstration

1. Log on to lnx201

   ```
   ssh kls286@lnx201.classe.cornell.edu
   ```

2. Change to a working directory

   ```
   cd /nfs/chess/user/kls286/demo
   ```

3. Use `emacs` to write a “helloworld” script

4. Change the file mode of the script to be executable by the user who owns it

   ```
   chmod u+x helloworld.sh
   ```

5. Hop to an a compute farm node for interactive jobs

   ```
   qrsh -q interactive.q
   ```

6. Change directories

   ```
   cd /nfs/chess/user/kls286/demo
   ```

7. Run the script

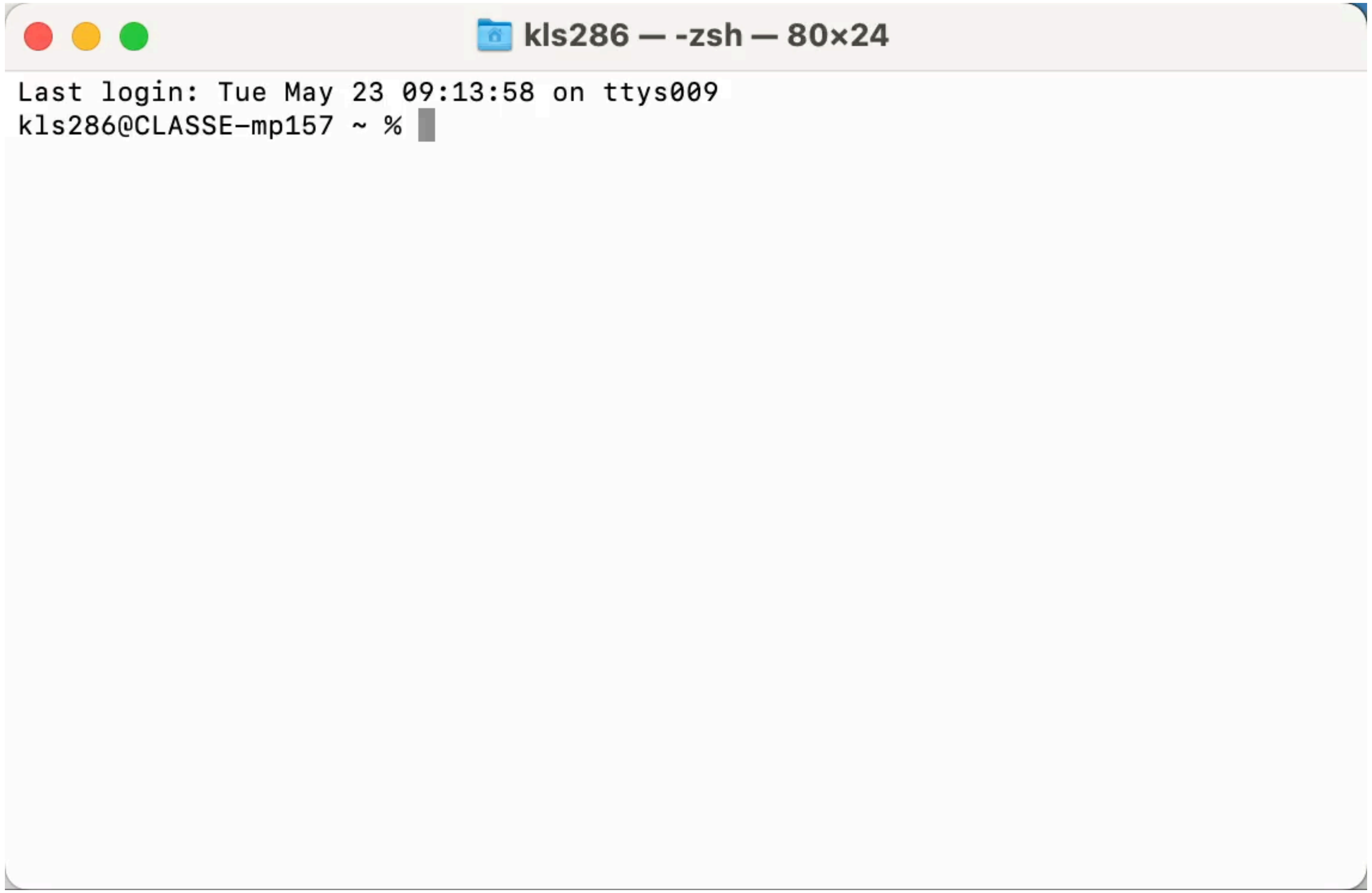
   ```
   ./helloworld.sh
   ```

8. Log out of the compute farm

   ```
   exit
   ```

9. Log out of lnx201

   ```
   exit
   ```



```
Last login: Tue May 23 09:13:58 on ttys009
kls286@CLASSE-mp157 ~ % 
```

Scientific Computing @ CLASSE

Developing python code

2024-06-12

Overview

- Managing python environments for development
- Introduction to jupyter
- Best practices: docstrings and style

Developing python code

Managing environments

System-wide default python

- No version options
- No permission to install packages like numpy, scipy, matplotlib, etc.
- Can’t keep track of the environment requirements for *your* project

Your own python environment

- Use any version of python
- Permission to install any packages your project needs
- Easy to keep track of the environment requirements for *your* project

Developing python code

Managing environments

`conda`

- Choose any python version
- Install packages, system libraries
- Install packages published for `conda` or `pip`
- Can be slow

`venv`

- Must use the system default python version
- Install only packages published for only `pip`
- Usually faster

Developing python code

Exercise 1

To use `conda`, install miniforge and activate the base environment. On a terminal on the CLASSE Linux system, run:

1. `cd /cdat/tem/<CLASSE-ID>`

2. `curl -L -O "https://github.com/conda-forge/miniforge/releases/latest/download/Miniforge3-Linux-x86_64.sh"`

3. `chmod u+x Miniforge3-Linux-x86_64.sh`

4. `./Miniforge3-Linux-x86_64.sh`

    a. Accept license agreement

    b. Specify new install directory: `/cdat/tem/<CLASSE-ID>/miniforge3`

    c. DO NOT allow the installer to update your shell profile to automatically initialize conda

5. `source miniforge3/bin/activate`

Developing python code

Exercise 2

Create and activate an environment that contains the latest version of python. In the same terminal used for exercise 1, run:

1. `which python`
2. `conda create -n myenv python`
   a. respond to “`Proceed ([y]/n)?`” prompt with “`y`”
3. `conda activate myenv`
4. `which python`
5. Note the different outputs from the 1st and 4th commands!

Developing python code

Exercise 3

Execute some python code in your new environment. In the same terminal used for the previous two exercises, run:

1.
   ```
   echo "print('hello world')" > helloworld.py
   ```

2.
   ```
   python helloworld.py
   ```

Developing python code

Exercise 4

Install a package in your new environment. In the same terminal used for the previous three exercises, run:

1. `python -c "import numpy"`

2. `conda install -y numpy`

3. `python -c "import numpy"`

Developing python code

https://jupyter01.classe.cornell.edu

- Another way to interact with the CLASSE filesystem
- Open a terminal, create / edit files, or use *jupyter notebooks*
- Notebooks can be a friendly option for developing python code if you're not comfortable using the terminal, but…
- There's a time and place for notebooks. Your project mentor can tell you if a jupyter notebook is an acceptable form for the final version of your code
- https://wiki.classe.cornell.edu/Computing/JupyterHub

Developing python code

https://jupyter01.classe.cornell.edu

- `jupyter01`'s file browser provides access to your `/home/<CLASSE-ID>` directory
- Recall: your work belongs somewhere in `/nfs/`…, NOT `/home/<CLASSE-ID>`
- Solution: use the symbolic link created in [an exercise from the previous section]() to navigate to an appropriate directory for your project files
- To make a python environment available for use in jupyter, one must install an "ipykernel" for it.

Developing python code

Exercise 4

Install an ipykernel for the environment you created in exercise 2

1. `pip install ipykernel`

2.
```
python -m ipykernel install --user --name=my-python-env
--display-name "My Python Env"
```

3. In https://jupyter01.classe.cornell.edu, open a new python notebook in select "My Python Env" for the kernel

Developing python code

Exercise 5

Write and run some code in your new python notebook. The code should print “`hello world`” when the cell is run.

Developing python code

Best practices — docstrings

- Docstring conventions — https://peps.python.org/pep-0257/
- Every module, class, and function in your project should have a docstring
- Docstrings should be written with python's built-in `help(object)` function and automatically-generated human-readable API documentation in mind
- Pick a canonical format for your docstrings and stick with it. If you don't already have one picked out, I recommend choosing the sphinx docstring format
- TIP: write docstrings for each module, class, and function BEFORE you write the actual code that goes inside
    - …but if you don't, ChatGPT usually does a good job at writing docstrings if you ask nicely

Developing python code

Docstrings & `help(obj)` example

```python
def func_no_docstring(x, y=None):
    # Describe what the function does.
    # describe what the argument `x` represents
    # describe what the argument `y` represents
    # describe what this function returns
    return None
```

```
> help(func_no_docstring)
Help on function func_no_docstring in module __main__:

func_no_docstring(x, y=None)
```

```python
def func_with_docstring(x, y=None):
    """Describe what the function does.

    :param x: describe what the argument `x` represents
    :type x: object
    :param y: describe what the argument `y` represents,
        defaults to None
    :type y: object, optional
    :return: describe what this function returns
    :rtype: object
    """
    return None
```

```
> help(func_with_docstring)
Help on function func_with_docstring in module __main__:

func_with_docstring(x, y=None)
    Describe what the function does.

    :param x: describe what the argument `x` represents
    :type x: object
    :param y: describe what the argument `y` represents,
        defaults to None
    :type y: object, optional
    :return: describe what this function returns
    :rtype: object
```

Developing python code

Best practices — style

- Style conventions — https://peps.python.org/pep-0008/
- Variable / function names: `snake_case`
- Class names: `CamelCase`
- Line widths: 79 or 99 characters (code) or 72 characters (comments & docstrings)
- …and many more guidelines that you are encouraged to follow
- You may break a guideline in PEP8 “when applying the guideline would make the code less readable, even for someone who is used to reading code that follows this PEP.”

Developing python code

Exercise 6

Copy the following code into a cell in your new jupyter notebook, then edit it so that it adheres to best practices:

```python
# Prompt: problem 1 from Project Euler https://projecteuler.net/problem=1
def Sum_MultiplesofThreeor_five_below(n):
    Result=0
    for I in range(n):
        if I%3 == 0 or I % 5==0:
            Result +=I
    return Result
print(Sum_MultiplesofThreeor_five_below(1000))
```

Scientific Computing @ CLASSE

Version control

2024-06-14

Overview

- What is version control
- What is git
- Where to host your project’s code repository
- Practice using git

Version control

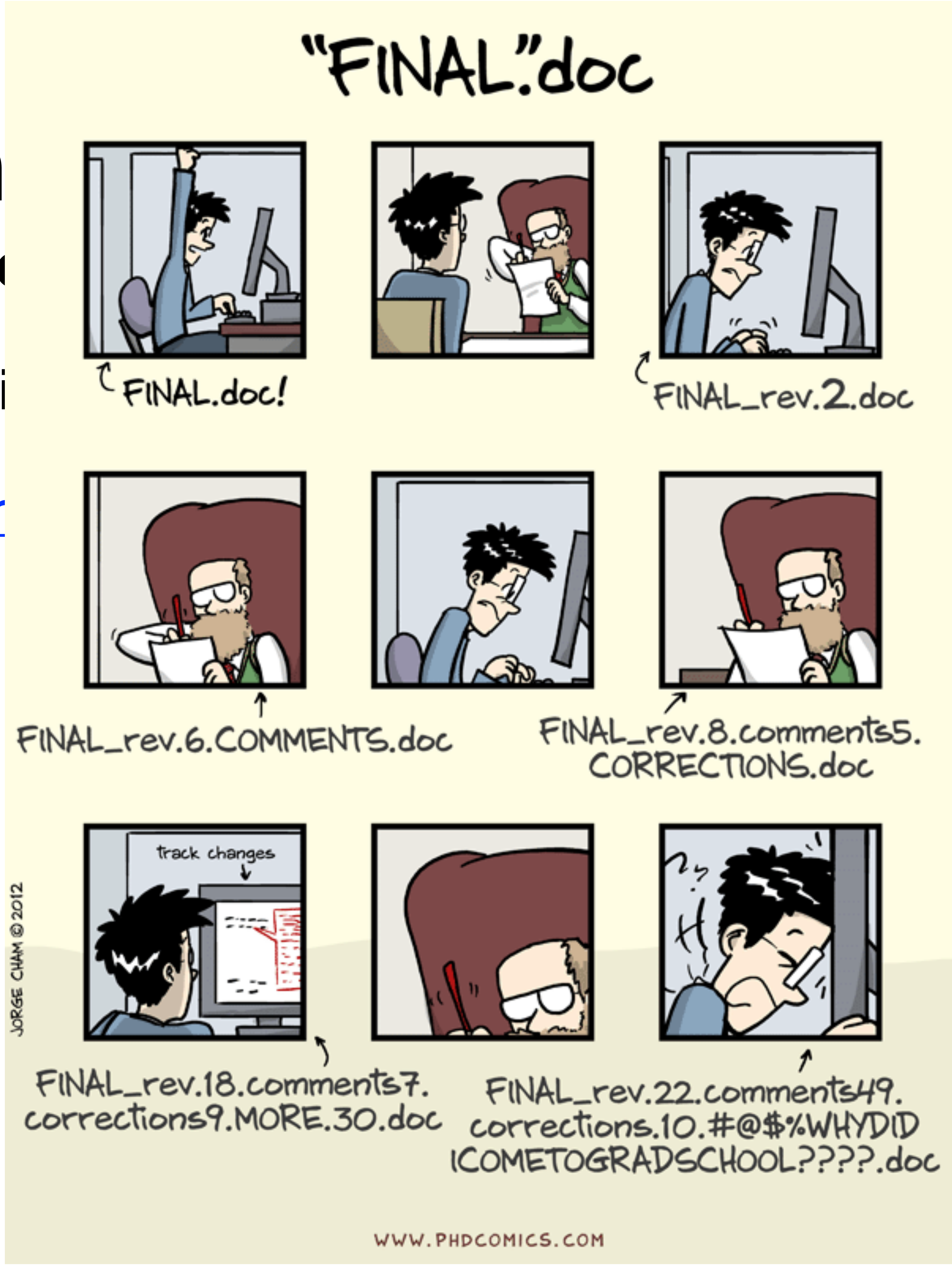
What is version control?

- Description of version control, and the motivation for having sophisticated tools to help us do it: https://xcitecourse.org/theme1/pe103/vcs#why-do-we-need-version-control

Version con

What is version c

- Description of versi ing sophisticated
  tools to help us do 03/vcs#why-do-we-
  need-version-contr



https://phdcomics.com/comics/archive.php?comicid=1531

Version control

What is git?

- git: the near-ubiquitous choice for version control software
- `git`: command-line tool for interacting with git repositories
- git repository: a copy of your project files and the history of changes you made to them. It helps you:
  - Preserve snapshots of your project at different stages of development
  - Develop different versions of your project that branch from a common root
  - Merge branching versions of your project back to a common root
- A git repository can be hosted remotely using tools like github or gitlab. These help you:
  - Share / distribute / deploy your project
  - Track issues
  - Publish documentation

Version control (git)

Where to host your repository

- CHESS students:
  - Your project’s git repository will be hosted in one of two places:
    - For projects that should be public:

      https://github.com/CHESSComputing
    - For projects that should be shared only within CLASSE:

      https://gitlab01.classe.cornell.edu
  - There is a “correct” place to host every one of your projects. Ask your mentor if you are unsure where your project belongs.
- All other students: ask your project mentor

Version control (git)

Vocabulary

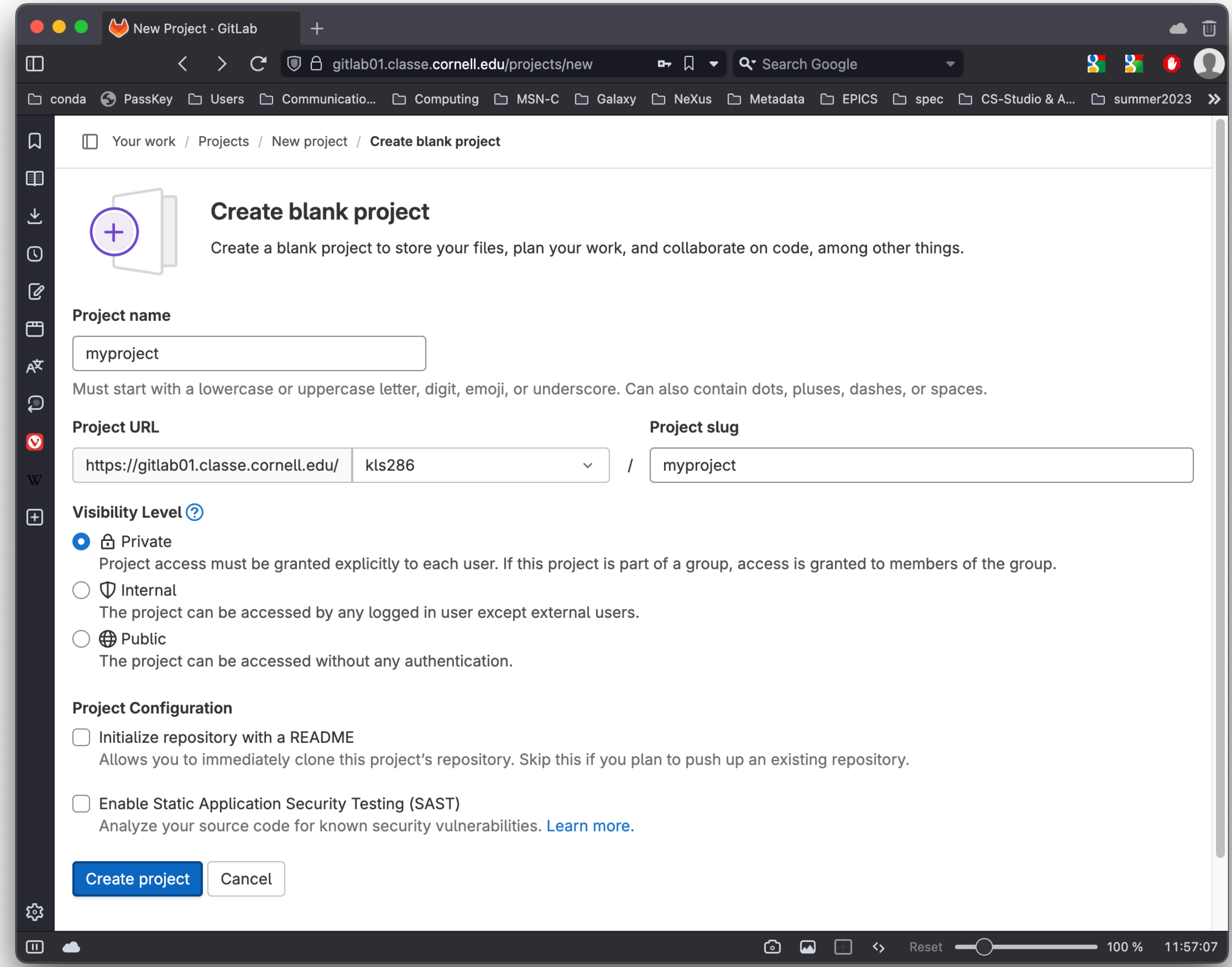
- **repository**: a collection of files with a history of developer-created checkpoints that preserve the state of those files at different stages (e.g. commits)
- **remote repository**: a repository accessible via URL (i.e. the version hosted on github/gitlab)
- **local repository**: a repository on your local filesystem
- **clone**: a local repository that is a copy of a remote repository
- **pull**: an act that updates your local repository with any new changes on the remote repository
- **commit**: an entry / checkpoint in the preserved history of a repository
- **push**: an act that updates a remote repository with changes (like commits) made on a local repository

Version control (git)

Exercise 1

Set up a blank repository:

1. Log on to https://gitlab01.classe.cornell.edu
2. Create a new repository for your project
    1. In upper righthand corner, click “New Project”
    2. Create a blank repository (see screenshot —>)
3. Run:
    1. `git clone <https://gitlab01.classe.cornell.edu/kls286/myproject.git>`
    2. `cd myproject`
    3. `git checkout -b main`



Your work / Projects / New project / **Create blank project**

Create blank project

Create a blank project to store your files, plan your work, and collaborate on code, among other things.

Project name

myproject

Must start with a lowercase or uppercase letter, digit, emoji, or underscore. Can also contain dots, pluses, dashes, or spaces.

Project URL: https://gitlab01.classe.cornell.edu/ kls286 / **Project slug**: myproject

Visibility Level

● Private
Project access must be granted explicitly to each user. If this project is part of a group, access is granted to members of the group.

○ Internal
The project can be accessed by any logged in user except external users.

○ Public
The project can be accessed without any authentication.

Project Configuration

□ Initialize repository with a README
Allows you to immediately clone this project’s repository. Skip this if you plan to push up an existing repository.

□ Enable Static Application Security Testing (SAST)
Analyze your source code for known security vulnerabilities. Learn more.

Create project | Cancel

Version control (git)

Basic development workflow

- At any point:
    - Use `git status` to examine the state of your local clone
    - Use `git diff` to examine the differences between files in the local clone and their “official” copy
- To make changes:
    1. Create / modify files in a local clone
    2. `git add` the files whose changes you want to keep
    3. `git commit` the additions
    4. (optional) repeat steps 1-3
    5. `git push` the commits

Version control (git)

Exercise 2

Practice committing & pushing a change to the repo created in Exercise 1. Run:

```
1. git status
2. touch README.md
3. git status
4. git add README.md
5. git status
6. git commit -m "add README"
7. git status
8. git push --set-upstream origin main
9. git status
```

Version control (git)

Exercise 3

Practice pulling changes to your local clone of the repo created in Exercise 1.

- In the gitlab web interface, edit README.md
- In the terminal, run:

```
1. git fetch

2. git status

3. git pull

4. git status
```

Version control (git)

commits

- Commit early and often so you can go back to previous versions of your project if needed

- Each commit should represent one incremental change to your project (e.g. don’t fix a bug *and* update the documentation in the same commit)

- Every commit needs a message that describes the changes you’re making

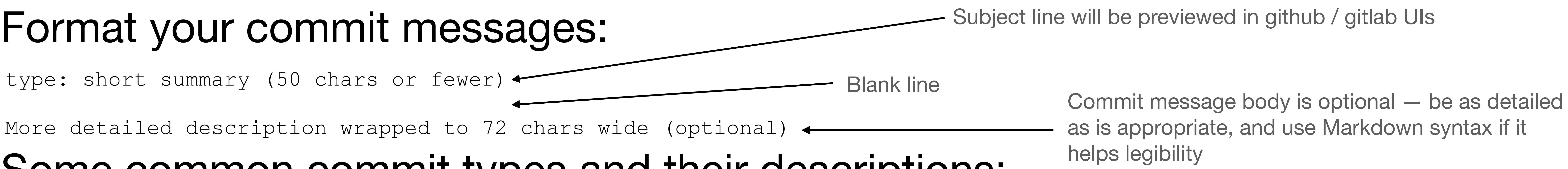
- To make the commit history easy to read, stick to conventions

Version control (git)

commit message conventions

- Use the imperative mood (e.g. “Fix bug”, NOT “Fixed bug” or “Fixes bug”)

- Format your commit messages:



```
type: short summary (50 chars or fewer)

More detailed description wrapped to 72 chars wide (optional)
```

Subject line will be previewed in github / gitlab UIs

Blank line

Commit message body is optional — be as detailed as is appropriate, and use Markdown syntax if it helps legibility

- Some common commit types and their descriptions:

```
feat: addition of some new features
add: changes to add new capability or functions
cut: removing the capability or functions
fix: a bug fix
bump: increasing the versions or dependency versions
build: changes to build system or external dependencies
make: change to the build process, or tooling, or infra
ci: changes to CI configuration files and scripts
doc: changes to the documentation
test: adding missing tests or correcting existing tests
chore: changes for housekeeping (avoiding this will force more meaningful message)
refactor: a code change that neither fixes a bug nor adds a feature
style: changes to the code that do not affect the meaning
perf: a code change that improves performance
revert: reverting an accidental commit
```

Version control (git)

Some security considerations

- Never keep passwords, API keys, or other “secrets” in regular files in any git repository
- Never keep CLASSE hostnames, addresses, or file paths in a git repository not hosted on gitlab01.classse.cornell.edu
- If you are unsure: ask your project mentor

End