While you're waiting, make sure you have a GitHub account! It's free at github.com.

EIEIOO 2021: Introduction to the Command Line (CLI) and *git*

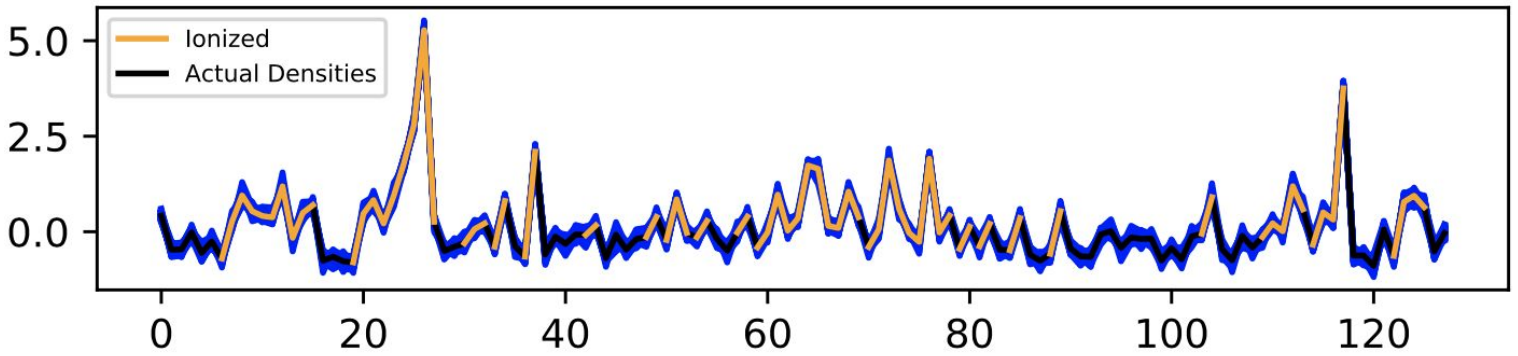
Presented by Sabrina Berger
with slides by Laurent Mackay
and Marcus Merryfield

sabrina.berger@mail.mcgill.ca
May 4, 2021

Most of these slides were made for the 2020 McGill Physics Hackathon Coding workshops by Marcus Merryfield (MM) & Laurent MacKay (LM).

About Me

- Completed my undergraduate degree at UC Berkeley before moving to Montréal
- 2nd year MSc physics student at McGill University
- Research interests are a mix between localizing fast radio bursts with CHIME Outriggers and placing constraints on reionization
- Here's a random plot I made yesterday:



Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Understand how to use *git*

- What is git?
- Tutorial 1: Set up your first git repository
- Tutorial 2: Make a branch
- Tutorial 3: Check out an old version of your code

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

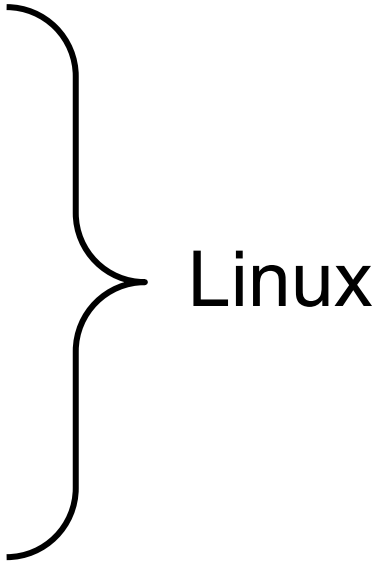
- **What is Linux?**
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Understand how to use *git*

- What is git?
- Tutorial 1: Set up your first git repository
- Tutorial 2: Make a branch
- Tutorial 3: Check out an old version of your code

What is Linux?

- Collection of Open Source Unix-like Operating Systems
  - Linux → Linux Is Not UNIX
- Unix Filesystem
  - Everything is a file (Everything)
  - Every file has a place. You don't have to put it there, but you should.
- Large library of software tools
- A shell scripting environment
  - Navigate the filesystem
  - Combine the software tools to accomplish complex tasks



What is Linux?

- Collection of open source unix-like operating systems
  - Linux → Linux Is Not UNIX

Linux Developer: Linus Torvalds

- Software Library
- Kernel
- UNIX Filesystem

Linux Distribution (Ubuntu, Fedora, Debian, Arch,...)

Why you should use Linux?

1. Its free and open source!
    a. Many flavors (distributions) to choose from, almost all of them are free.

2. When it works, it works!
    a. Can be left running for months/years without any issues.
    b. No need to reboot after installing software!

The Most Useful Linux Troubleshooting Advice

Google:
Error Message/file + [Linux Distribution Name]

Check out the top 2-3 results, see what they agree on.

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- **Tutorial 1: Navigating the Filesystem**
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Understand how to use *git*

- What is git?
- Tutorial 1: Set up your first git repository
- Tutorial 2: Make a branch
- Tutorial 3: Check out an old version of your code

Tutorial 1: Navigating the Filesystem

1. Open Terminal (Mac) or Ctrl + Alt +T (Windows/Ubuntu)
   a. It may look like nothing, but it can do almost everything.
2. Check out your home folder

   ls
   (Lists the contents of your home folder)

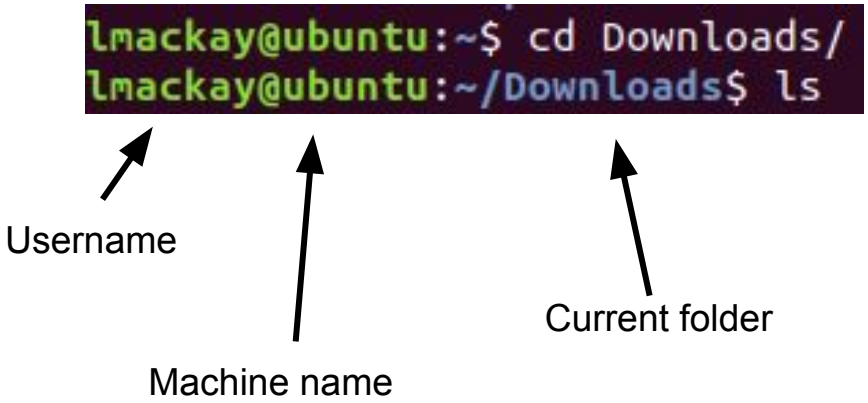
   ls -a
   (Includes “hidden” files, i.e., files whose name starts with ‘.’)

   (Files and Directories will probably be colored differently)

Tutorial 1: Navigating the Filesystem

1. Move into a new directory (e.g. Downloads)

   cd Downloads



Tutorial 1: Navigating the Filesystem

1. Move into a new directory (e.g. Downloads)
   cd Downloads

   ```
   lmackay@ubuntu:~$ cd Downloads/
   lmackay@ubuntu:~/Downloads$ ls
   ```
2. Is there anything there?
   ls
   (maybe not)
   ls -a
   (you will notice '.' and '..')
3. What does .. mean?
   cd ..
4. pwd (what is the present working directory?)

Linux Shorthand:

. → Current Directory

.. → Parent Director

~ → Home Folder

Tutorial 1: Navigating the Filesystem

1. Try using an absolute file path
   `cd /home/[username]/Downloads`

2. You can use ~ as a shortcut for your home folder
   `cd ~/Documents`

Let's start playing around with files:

1. Copy a file into your Documents folder (commands are equivalent)
   absolute paths: `cp /home/[username]/.bashrc /home/[username]/Documents/`
   relative paths: `cp ../.bashrc ./`

2. You can copy a file to a new filename
   `cp ~/.bashrc ./dummyfile`

Tutorial 1: Navigating the Filesystem

1. Make a new directory:
   `mkdir dummydir`

2. Move your dummyfile into dummydir:
   `mv dummyfile dummydir`

3. Check that is worked:
   `ls dummyfile`
   (should return ls: cannot access 'dummyfile': No such file or directory)

   `ls dummyfolder/dummyfile`
   (should not give an error)

Tutorial 1: Navigating the Filesystem

Let's make a sample Python file with ***vim (also check out nano and emacs)***.

1. Open your file (notice the .py ending): `vim test.py`
2. Press `i` to enter insert mode
3. Press `ESC` to enter
    a. Force quit: `:q!`
    b. Quit and write: `:wq`
    c. Force quit and write: `:wq!`
4. Run your python file: `python test.py`

Tutorial 1: Navigating the Filesystem

Let’s clean up after ourselves

1. `rm ./.bashrc`
   (deletes the .bashrc file we copied into Documents)

2. `rm -r dummydir`
   (recursively deletes anything inside dummydir and then deletes dummydir)

Tutorial 1: Navigating the Filesystem

Summary of Commands:

1. `ls` lists contents of current directory
2. `ls /path/to/dir` lists contents of a specific directory
3. `cd /path/to/dir` changes current directory
4. `cp` copies files
5. `mv` moves/renames files/directories
6. `mkdir` makes directory
7. `rm` deletes files/directories

Specific options for all of these commands can be found using the `--help` flag

More detailed instructions can be found using `man` (e.g., try `man ls` or `man echo`)

Activity 1: Navigating the Filesystem

1. Make two directories: *testing* and *EIEIOO_Scripts*
2. Make two sample python scripts in the *testing* directory
3. Move one script to *EIEIOO_Scripts*
4. Copy the other script to *EIEIOO_Scripts*
5. Delete the *testing* folder
6. Run your python scripts

```
physuser@b77ad2eaee59:~/EIEIOO_Scripts$ ls
test.py  test2.py
```

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- **Tutorial 2: File Permissions**
- Tutorial 3: The Builtin Software Library

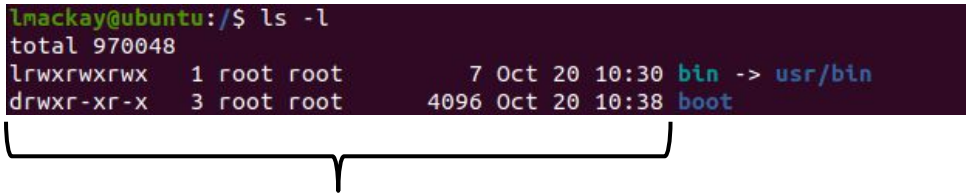
Part 2: Understand how to use *git*

- What is git?
- Tutorial 1: Set up your first git repository
- Tutorial 2: Make a branch
- Tutorial 3: Check out an old version of your code

Tutorial 2: File Permissions

Lets have a look at the root folder again:

`ls -l /`



What is all this stuff?

Tutorial 2: The Linux Filesystem

```
lmackay@ubuntu:/$ ls -l
total 970048
lrwxrwxrwx   1 root root          7 Oct 20 10:30 bin -> usr/bin
drwxr-xr-x   3 root root       4096 Oct 20 10:38 boot
```

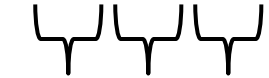
The first character tells you what type of file: - (regular file), d (directory), l (link)

Tutorial 2: File Permissions

The next nine characters gives you three different sets of “permissions” for the file

- Three different levels of control over the file

```
lmackay@ubuntu:/$ ls -l
total 970048
lrwxrwxrwx   1 root root          7 Oct 20 10:30 bin -> usr/bin
drwxr-xr-x   3 root root       4096 Oct 20 10:38 boot
```



Owner

Group

Everyone Else

r = Reading Permitted

w = Writing Permitted

x = Executing Permitted

Tutorial 2: File Permissions

```
lmackay@ubuntu:/$ ls -l
total 970048
lrwxrwxrwx   1 root root          7 Oct 20 10:30 bin -> usr/bin
drwxr-xr-x   3 root root       4096 Oct 20 10:38 boot
```

Number of links/directories inside a link/directory

Tutorial 2: File Permissions

```
lmackay@ubuntu:/$ ls -l
total 970048
lrwxrwxrwx   1 root root          7 Oct 20 10:30 bin -> usr/bin
drwxr-xr-x   3 root root       4096 Oct 20 10:38 boot
```

Owner of the file

You can modify who owns a file with: chown [username] [filename]

Tutorial 2: File Permissions

```
lmackay@ubuntu:/$ ls -l
total 970048
lrwxrwxrwx   1 root root          7 Oct 20 10:30 bin -> usr/bin
drwxr-xr-x   3 root root       4096 Oct 20 10:38 boot
```

Group that owns the file

You can add users to a group using: usermod -aG [groupname] [username]

Tutorial 2: File Permissions

```
lmackay@ubuntu:/$ ls -l
total 970048
lrwxrwxrwx   1 root root          7 Oct 20 10:30 bin -> usr/bin
drwxr-xr-x   3 root root       4096 Oct 20 10:38 boot
```

Size of the file in bytes. Try using `ls -lh` for human readable sizes.

Tutorial 2: File Permissions

```
lmackay@ubuntu:/$ ls -l
total 970048
lrwxrwxrwx   1 root root          7 Oct 20 10:30 bin -> usr/bin
drwxr-xr-x   3 root root       4096 Oct 20 10:38 boot
```

Date of last modification

Tutorial 2: File Permissions

```
lmackay@ubuntu:/$ ls -l
total 970048
lrwxrwxrwx   1 root root          7 Oct 20 10:30 bin -> usr/bin
drwxr-xr-x   3 root root       4096 Oct 20 10:38 boot
```

Filename

Tutorial 2: File Permissions

Ok so we have seen how to give ourselves ownership (`chown`) or group membership (`usermod -aG`).

What about everyone else? Can we modify the owner/group permissions?

`chmod u=rwx,g=rx,o=r [filename]`

u = user = owner

g = group

o = other = everyone else

Everyone can read, write, and execute: `chmod 777 <filename>`

Tutorial 2: File Permissions

Ok so we have seen how to give ourselves ownership (chown) or group membership (usermod -aG).

What about everyone else? Can we modify the owner/group permissions?

```
chmod u=rwx,g=rx,o=r [filename]
```

u = user = owner

g = group

o = other = everyone else

Different Permutations:

0 – no permission

1 – execute

2 – write

3 – write and execute

4 – read

5 – read and execute

6 – read and write

7 – read, write, and execute

Tutorial 2: File Permissions

The dangerous but useful: **chmod 777 <filename>**

Everyone can read, write, and execute this file.

Exercise 2: File Permissions

1. Change one of your sample Python files permissions so that you get the following error when you try to run it:

   ```
   python: can't open file 'test.py': [Errno 13] Permission denied
   ```

2. Figure out how to successfully run it again

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- **Tutorial 3: The Builtin Software Library**

Part 2: Understand how to use *git*

- What is git?
- Tutorial 1: Set up your first git repository
- Tutorial 2: Make a branch
- Tutorial 3: Check out an old version of your code

Tutorial 3: The Builtin Software Library

Super random question:

How many times are empty lists initialized as

```
my_list = list()
```

versus

```
my_list = []
```

in the base python library on your system?

Tutorial 3: The Builtin Software Library

1. Let’s find where python is:
   `whereis python`
   For me, there are multiple python versions installed.

2. `cd /usr/lib/python3`

   Let’s check out the Python versions installed here.

Tutorial 3: The Builtin Software Library

1. Lets see how many python files there are
   `find . -name '*.py'`

   Look ‘here’ for files whose name matches the pattern *.py (*=wildcard)
   (Ok, that’s a lot of files!)

2. `grep -r --include '*.py' '= list()'`

   Recurse down the filesystem, looking inside files that end with .py searching for ‘= list()’

Tutorial 3: The Builtin Software Library

1. Now to count them up
   `grep -r --include '*.py' '= list()'  |  wc -l`

   The character | “pipes” the output of `grep` into `wc` which counts the number of words it is (or in this case lines due to `-l`)

2. Ok now lets try to repeat that with my_list = []
   `grep -r --include '*.py' '= []' | wc -l`

   You will get an error:
   The ‘[]’ characters are special and must be “escaped”
   `grep -r --include '*.py' '= \[\]' | wc -l`

Tutorial 3: The Builtin Software Library

Great!

So how many instances did you find?

More generally, grep is extremely useful for finding strings (or a RegExp) in text files when you can’t remember which file its in.

Tutorial 3: The Builtin Software Library

Some commands I commonly use:

find: find files
grep: search for strings/patterns inside text files
top or htop (fancy version): similar to task manager
df -h: check how much free disk space is available on your system

Tip: Google “How to do <blank> command line linux”

Exercise 3: The Builtin Software Library

Determine how many times ‘import’ is used in your Python3 installation. For me, it’s 18!

Lastly, what about a job I need to quit?

Ok now we started a rogue process which will never finish and will eat up our hard drive. What do we do?

Use the pid to kill the process!

`kill 25879`

If you do not remember the pid, use `jobs -l` to find the jobs currently running in a given shell.

```
n$ jobs -l
[1]+ 25879 Running                 while sleep 1; do
    date >> curr_time;
done &
```

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Understand how to use *git*

- **What is git?**
- Tutorial 1: Set up your first git repository
- Tutorial 2: Make a branch
- Tutorial 3: Check out an old version of your code

What *is* Git?

- Git is what is known a Version Control System (VCS)
- By using git, you can keep track of ***what*** changed in a coding project, ***when*** it changed, ***who*** changed it, and ***why*** (if you're keeping good commit messages!)
- Particularly useful in collaborative projects, where multiple people are making changes at once. If anything in your changes conflicts (known as "***merge conflicts***"), changes can be made (sort of) gracefully
- In some ways, git is like a philosophy of how collaboration in code should be done

GitHub: working collaboratively online

- GitHub is an online service for hosting projects managed with git online
- Expands on the branching of git with useful collaborative features
  - ***Issues***
  - ***Pull requests***
  - ***Wikis, projects, …***

Note: *GitHub != git!*

- Very common misunderstanding
- *git* is the original VCS code, and doesn't have online hosting on its own
- *GitHub* is an online cloud hosting service with extra features for git projects

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

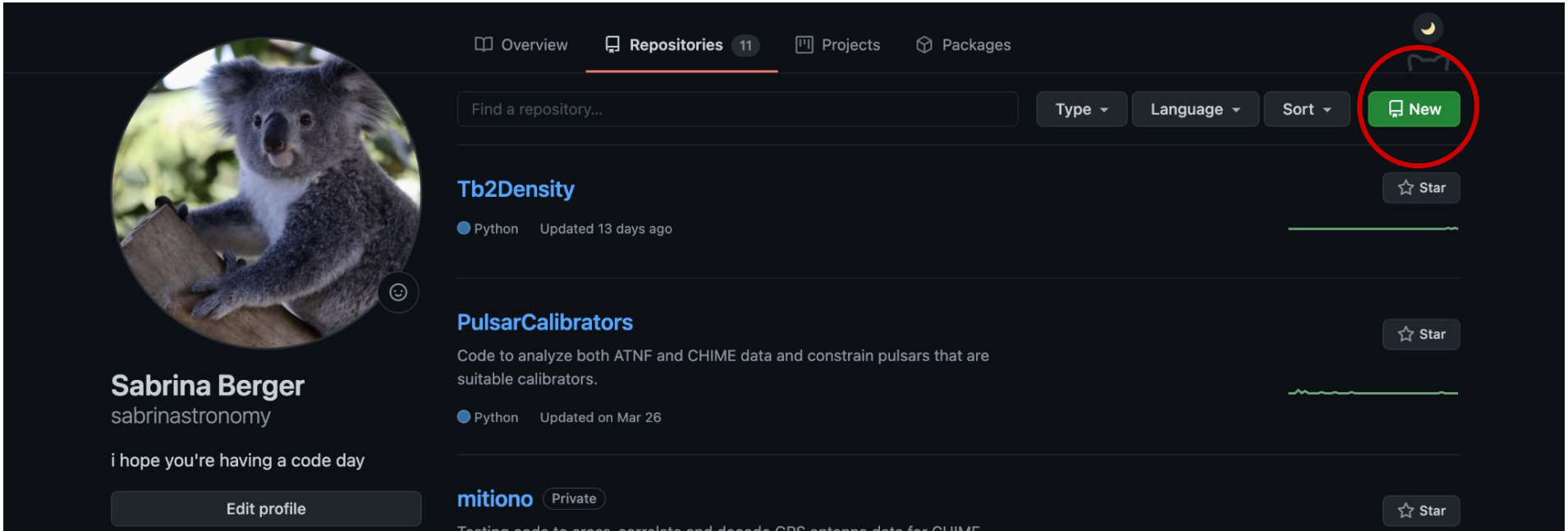
- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Understand how to use *git*

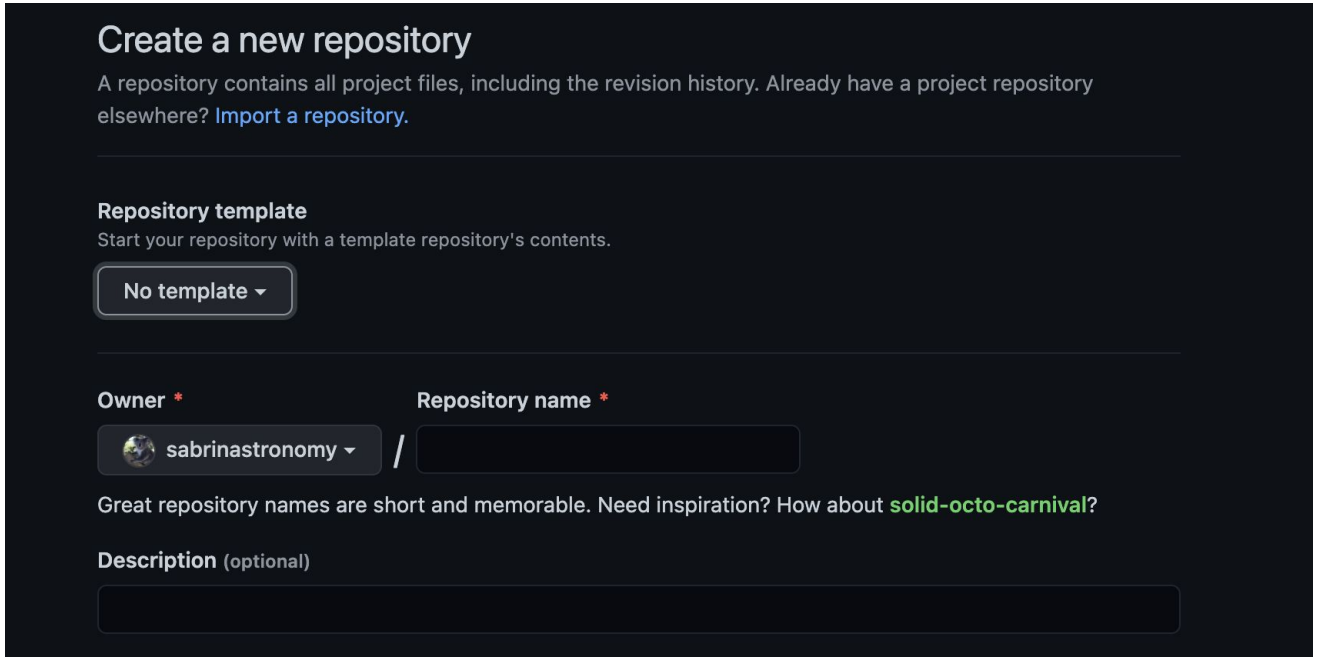
- What is git?
- **Tutorial 1: Set up your first git repository**
- Tutorial 2: Make a branch
- Tutorial 3: Check out an old version of your code

Tutorial 1: Making a repository

- Hopefully everyone's made a GitHub account to follow along with this tutorial -- if you haven't, you can quickly to follow along!



Follow the instructions to create a repository (don’t worry about any of the extra features for now).



Now we want to *git clone* the repository we just made on to our local machines (wherever we’d like!)

Tutorial 1: Making a repository (Summary)

- We can ***clone*** (*git clone*) GitHub repos to our local machine, which will copy over all the git history from the online project (**./git** folder!)
    - **Note:** this makes a ***remote branch*** called ***origin*** which is attached to the cloud, this can be a confusing detail if you accidentally make a local branch and then try and merge it into origin
- We can also create a git repository locally (*git init*), which creates a **./git** folder locally. We can then push to GitHub later
    - IMO, this workflow is confusing, and I recommend starting *all* personal coding projects with repositories on GitHub

Important basic git commands

- git --help
  - Gives a helpful list of git commands! Can also do `git {command} --help` for specifics on commands
- git config --global user.name “{username}”
- git config --global user.email {email}
  - **Note:** these are *git* associated names/emails, so they don’t have much to do with GitHub! Just useful for identifying yourself in local git projects
- git status
  - Shows you current changes
- git log
  - Shows you the commit history for your project
- git fetch
  - Updates information stored in the local **./git** folder, such as new remote branches

Making our first commit

- The most basic git workflow consists of three important steps:
  a. Use `**git add {file}**` to “add” a new file, or to “add” changes to an already existing file
     - You can use “wildcard” operators with this! E.g. `**git add .*py**` to add .py file changes
  b. Once you’re happy with your additions, use `**git commit -m “{useful message}”**` to add a commit with a helpful commit message explaining your changes
  c. Finally, we need to push our changes to the cloud. We’ll do this with the command `**git push origin master**`
     - Note that the ***origin*** here specifies the remote cloud, and ***master*** is the branch we’re committing to. By default, GitHub repos start with only a master branch

Pulling from remote

- To pull in any changes from collaborators, just use `**git pull**` in the relevant directory
    - This won't do anything for us now, since we just made this repo for ourselves... But it's VERY important to pull the most recent version of the repository before you start making changes!
    - Ideally if everyone was working on their own branches and being responsible about workflow, this wouldn't be an issue... But nobody's perfect :)

Summary

- Step 0: PULL changes that might have been made by collaborators
- Step 1: ADD our changes
- Step 2: COMMIT changes with a message (-m)
- Step 3: PUSH changes to the cloud
- **NOTE:** all the while we can check the STATUS of our additions!

Exercise 1:

1. Create a GitHub repository on your account
2. Move the files from *EIEIOO_Scripts* to the cloned repository and make your first commit.
3. Then change test2.py to print “One day Github will save me from the coding monsters”.
4. Push your changes to GitHub

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

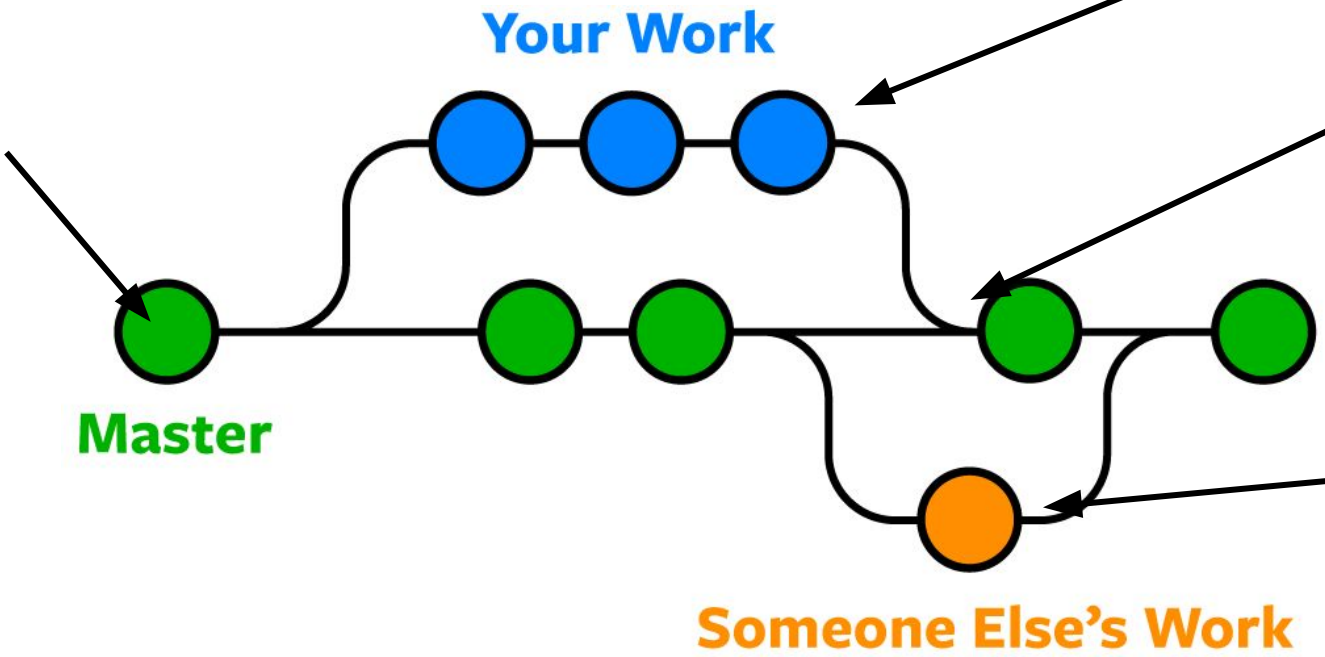
Part 2: Understand how to use *git*

- What is git?
- Tutorial 1: Set up your first git repository
- **Tutorial 2: Make a branch**
- Tutorial 3: Check out an old version of your code

Tutorial 2: Make a Git branch

You want to code a new feature in the code without breaking the stable code base, so you make your own branch!

Start with some original version of the code base



Merge your changes into the master branch once they're stable!

Others can write features in tandem!

Making our own branch

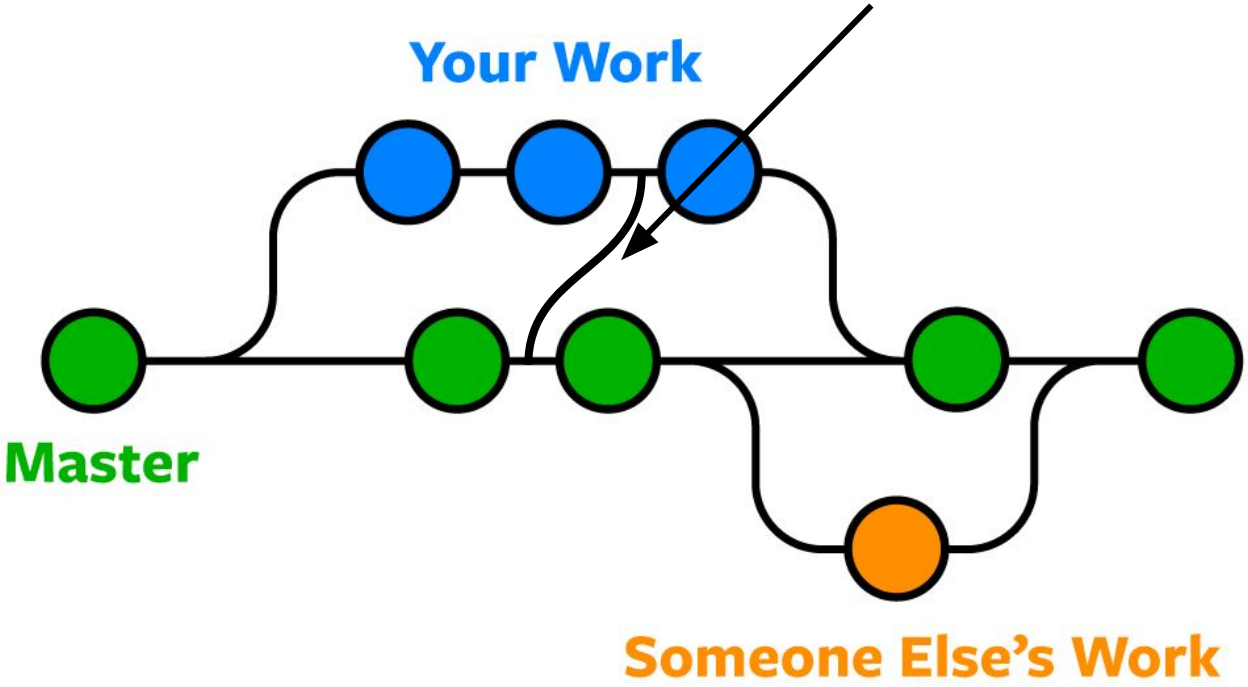
- We can take a look at the available branches with `**git branch**`
  - Can do `**git branch -r**` for remote branches, `**git branch -a**` for all branches
- Make our own branch with `**git branch {branch name}**`

- NOTE: `**git branch {branch name}**` only makes a branch *locally!* Later, we'll see how to get this branch on GitHub
- The branch command only *made* the branch. Now if we want to ***checkout*** to our new branch -- i.e. move to the space where we'll make our changes
    - `**git checkout {branch name}**`
    - **NOTE:** we can checkout to a new branch in one command! `**git checkout -b {new branch}**`
- Now we can safely make our changes without interfering in the stable master branch!
- Once we've ADDED and COMMITTED our changes, we can PUSH!
    - **NOTE:** we have to do `**git push --set-upstream origin {our branch}**` -- this is because our branch has only been local, until now: we're making our branch sync with the cloud with **--set-upstream**

Merging in changes from other branches

- Usually a good practice to compare differences between branches first
  - `**git diff {one-branch} {other-branch}**` to compare
- Now, say we're working on our own feature branch, and there are some useful changes on another part of the code base in ***origin/master*** we want on our branch
  - First: `**git pull**` to update your local branches with changes from the cloud (**origin**)
  - Next: `**git merge {other-branch}**` to put those changes in your current branch!

Merging in relevant changes from the master branch that might be useful for our testing (**note**: conflicts could arise!)

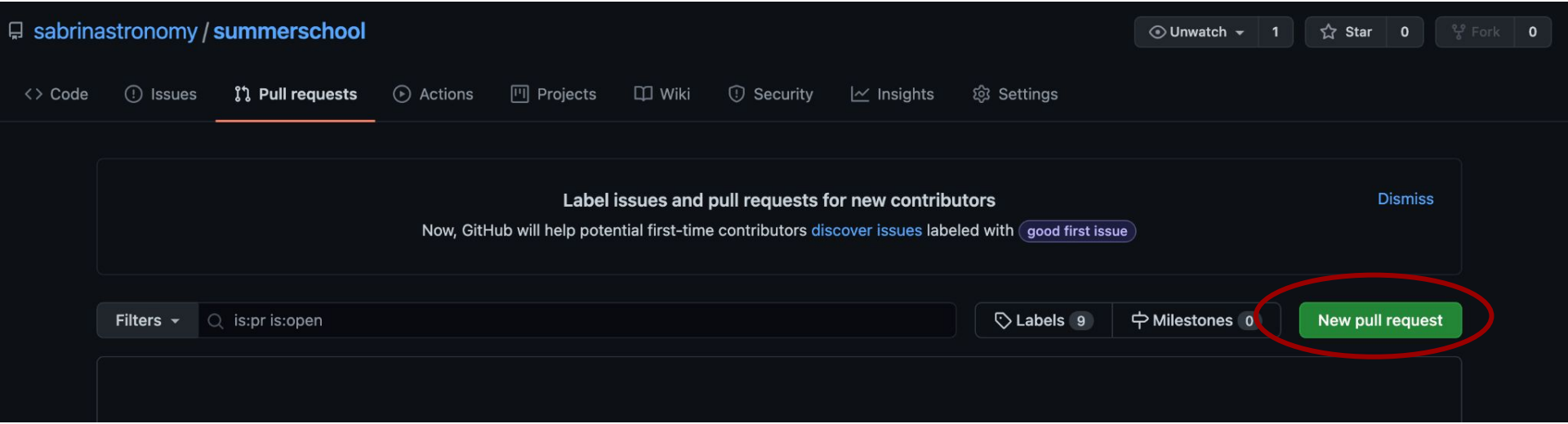


Exercise 2: Make your own branch

Create and push a new branch to your online GitHub repository for EIEIOO_Scripts. Call the branch “trying_new_things”.

Making a pull request

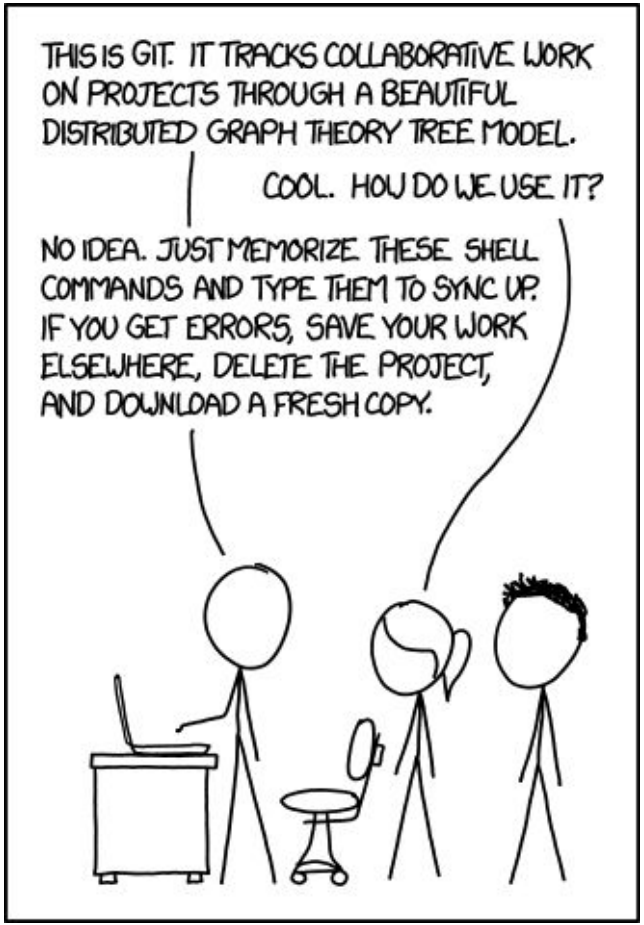
- When you're working in a collaboration, and you're ready to incorporate your changes into the master branch, you can make a pull request!



- Pull requests are a super useful way of keeping major code changes organized
- Rule of thumb: **master** branch should ALWAYS be deployable
    - This is why pull requests exist: typically if you're a part of a collaboration, there will be other people working on the code base with you. Usually there'll be one/a few people who manage most of a given repository, and making a pull request allows you to give them a chance to view your code, review it, suggest changes, and then finally accept the merge into master once it's deemed ready
- Workflow goes something like
    - Propose a new feature
    - Checkout a new branch to start working on your feature (make sure nobody interferes with your work so you don't get merge conflicts)
    - Keep pushing changes to your branch until things are stable/finished, then make a pull request

Dealing with merge conflicts

- Despite best efforts to keep organized, issues *will* arise!
- Merge conflicts are the part of git that will, at some point in your coding life, make you scream at your computer
- We can fix these problems pretty easily, in fact! Try not to resort to saving your changes locally, and re-downloading the whole repo



Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Understand how to use *git*

- What is git?
- Tutorial 1: Set up your first git repository
- Tutorial 2: Make a branch
- **Tutorial 3 (Final): Checkout an old version of your code**

Going back to an old commit when your new code breaks!

- You can get a log of all previous commits with
  - `**git log**`
- This should return your previous commits along with their corresponding hash, e.g.,

```
(base) sabrinaberger@sabrinastronomy summerschool % git log
commit 5d29d753054e787005f0202364e286d1211032db (HEAD)
```

- You can revert to a previous commit with
  - `**git checkout <commit hash> .** `

```
$ cat merge.txt
<<<<<<< HEAD
this is some content to mess with
content to append
=======
totally different content to merge later
>>>>>>> new_branch_to_merge_later
```

The important parts of a merge conflict will show up in our conflicted files:

- <<<<<<< HEAD
- =======
- >>>>>>> new_branch_to_merge_later

Think of the “=======” as the conflict divider. The content between HEAD and the divider is our content, and the content between the divider and the new_branch_to_merge_later is the content we tried to merge in. By reconciling the differences on these lines of code in a text editor, once you’re happy with the outcome, you can add/commit/push as usual!

Final Exercise

Restore your local *EIEIOO_Scripts* to its original commit before you modified the file.

Great job on all the tutorials today!

Resources

- How to install git on any OS
- A nice ELI5 git series
- "What is git" from Atlassian
- Basic git tutorial
- Reference for adding local git projects to the cloud
- An in-depth summary of remote branches
- Tutorial on how to deal with merge conflicts

Thank you for listening! And thanks again to Marcus (git) and Laurent (Unix/Linux/CLI) for the slides.

MCGILL PHYSICS

HACKATHON

See you in the fall at the McGill Physics Hackathon?

Questions?

You can also email me after at sabrina.berger@mail.mcgill.ca

Tutorial extra: Networked Computing

Open a secure connection to a shell instance on a remote machine:

`ssh [username]@[remote machine]`

e.g., `lmacka3@mimi.cs.mcgill.ca` (register here)

[Do what you need to do on the remote machine]

Close the shell:

`exit`

Tutorial extra: Networked Computing - Copying Files

You may need some of your files on remote machine. Use `scp [source] [target]`!

Lets make a dummy file: `touch ~/foo`

Copy the file to a remote machine:

```
scp ~/foo lmacka3@mimi.cs.mcgill.ca:/home/cnd/lmacka3
```

You may need to use ssh to figure out the remote path you are copying to.

You can also use rsync to transfer only new files

Demonstration extra: Networked Computing - Copying Files

I will now share my terminal to demonstrate copying some files on Compute Canada's Beluga.