A Git Tutorial

There is an excellent free on-line book on git:

https://git-scm.com/book/

…or do a web search on "git book".

Everything you'd want to learn about git is there.

That's it

We're done

The End

Unfortunately…

I was told I had to teach you something

A practical approach to git

based on interrogative pronouns

- Why
  - … do we use a version-control system?
- What
  - … are branches?
- Where
  - … should a "git repository" be?
- How
  - … does git track changes?
- When
  - … do we use tags?

Why use a version-control system?

When you edit a document normally, you "Save" once in a while.
You do this to protect work in volatile memory by moving it to non-volatile storage.
Call this "progress-based saving".
You can revert a change using "Undo", up to a point.

```
Analyze.py
```

Assume you wish save your document as of a specific point, after a completing a "task".
You define what a "task" is: adding a section; updating functionality; the work you've done today.
Call this "task-based saving".

```
Analyze.py
AnalyzeHistograms.py
Analyze-write-histograms.py
Analyze-write-histograms-and-ntuple.py
Analyze-derive-pt.py
```

Perhaps you want to keep track of your tasks. Or perhaps you want to go back to a prior point in the development of your file, so you can begin a related task from that point. One way to do this is to save multiple copies of your file with different names. This can become awkward.

One way to think of git: It offers an automated way to manage "task-based saving" for you.

Basic version control

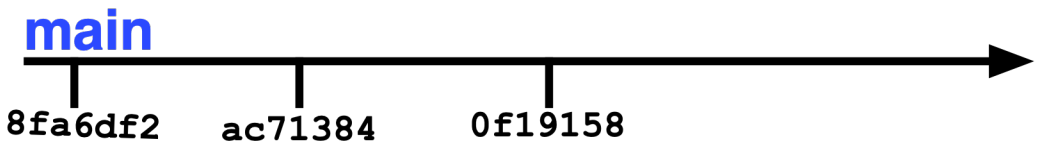
`git init -b main`	Set up the current directory for git. Call the principal file version 'main'. Only do this once.
`git add Analyze.C`	Include 'Analyze.C' to the list of files that git manages in this directory. Only do this once per file.
`git commit -a`	Save the current state of all the git-managed files in this directory. You will be prompted for a comment to document why you made this save.
`git commit -a -m "my comment"`	Supply the comment on the command line, to save a bit of time. **← This is the "bread-and-butter" git command. Feel free to "commit" a lot. The up-arrow is your friend.**
`git log`	Show a list of the commits you've made, along with comments
`git reflog`	Shorter list than 'git log', often more convenient.
`git revert HEAD`	"I don’t like any of the changes I made since my last commit. Undo all of them."
`git checkout 62ceb81`	"I looked in 'git reflog' for the ID number of an earlier commit. I want to go back and look at it."
`git merge main`	"I made changes based on that earlier commit, and I now want them to be my 'official' changes, replacing any later work."

`git reflog` example

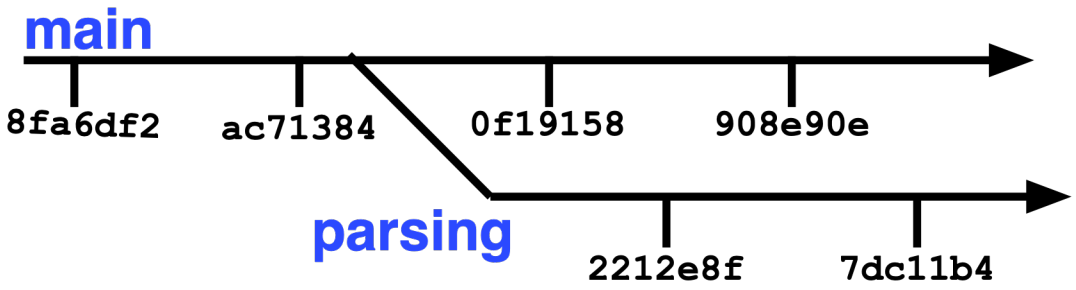
From one of my projects. Note how each commit has a unique ID. Also note that git tracks more than just "commits"; it includes all operational tasks in its log.

```
2aec340 HEAD@{32}: commit: Include a size() method in MCTrack
9a781a1 HEAD@{33}: commit: Clarify --uimacrofile
eaf6099 HEAD@{34}: commit: Move libUtilities to main build directory
f19e6e6 (linkdef) HEAD@{35}: merge linkdef: Fast-forward (no commit created; -m option ignored)
19a3b38 HEAD@{36}: checkout: moving from linkdef to include
f19e6e6 (linkdef) HEAD@{37}: checkout: moving from include to linkdef
19a3b38 HEAD@{38}: checkout: moving from linkdef to include
f19e6e6 (linkdef) HEAD@{39}: commit: Fancier trajectory accessors within MCTrack
ece7323 HEAD@{40}: commit: More debug output in GramsDetSim
d83a59e HEAD@{41}: commit: Increase cluster ID across all the hits in an event
2092911 HEAD@{42}: commit: Setting splitlevel=0 on all branches
a816069 HEAD@{43}: merge develop: Fast-forward
fc49857 HEAD@{44}: checkout: moving from develop to linkdef
a816069 HEAD@{45}: commit: Document changes from the 'include' branch
19a3b38 HEAD@{46}: merge include: Fast-forward
fc49857 HEAD@{47}: checkout: moving from include to develop
19a3b38 HEAD@{48}: commit: Revert documentation: all input energy is kinetic, not total
9c2df3d HEAD@{49}: commit: Move remaining images to sub-directories
f5ee6fa HEAD@{50}: commit: Move GramsSky images to sub-directory
14a2ec1 HEAD@{51}: commit: GramsSky energy now interpreted as kinetic energy by GramsG4
519bc04 HEAD@{52}: checkout: moving from linkdef to include
```

What are branches?



Suppose you're in the midst of your main task.
You make commits from time-to-time.



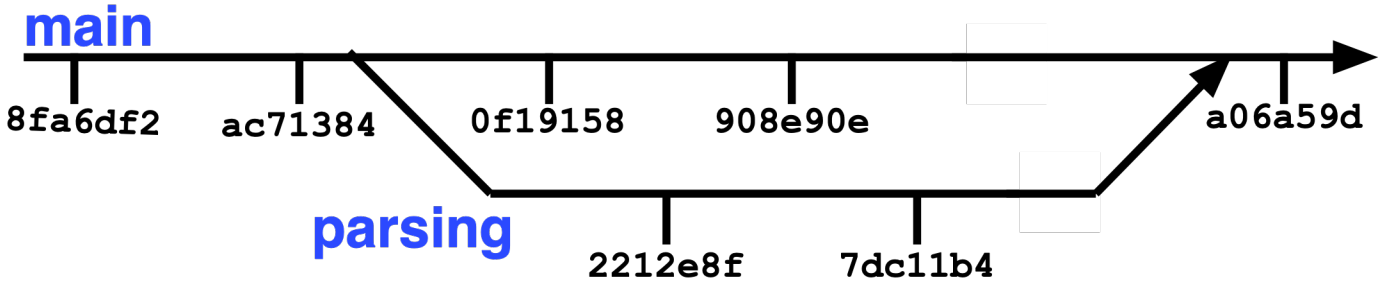
You're given a new task that's independent of your current main work, and you don't want to make two sets of changes at once. Instead, you create a new independent "branch" of your files for this task. The branch is named 'parsing' in this example.

```
git commit -a "save main work" # Before branching: Save your work
git reflog # to find the version ID you want
git checkout ac71384

git branch parsing # Create the new branch, named "parsing"
git checkout parsing # Start working in that new branch
emacs whatever.txt
git commit -a -m "a change in the 'parsing' branch
```

Getting a new task in the middle of other work: Does this really happen? All the time! Branching is especially useful when you're collaborating with others, each with their own tasks related to some "parent" set of files.

Merging branches

You've finished with the 'parsing' task. You want to merge the changes you made for that task with those you've continued to make in the main task. Git can automate this to some extent, but you may have to manually intervene if changes have been made to the same files in both branches.



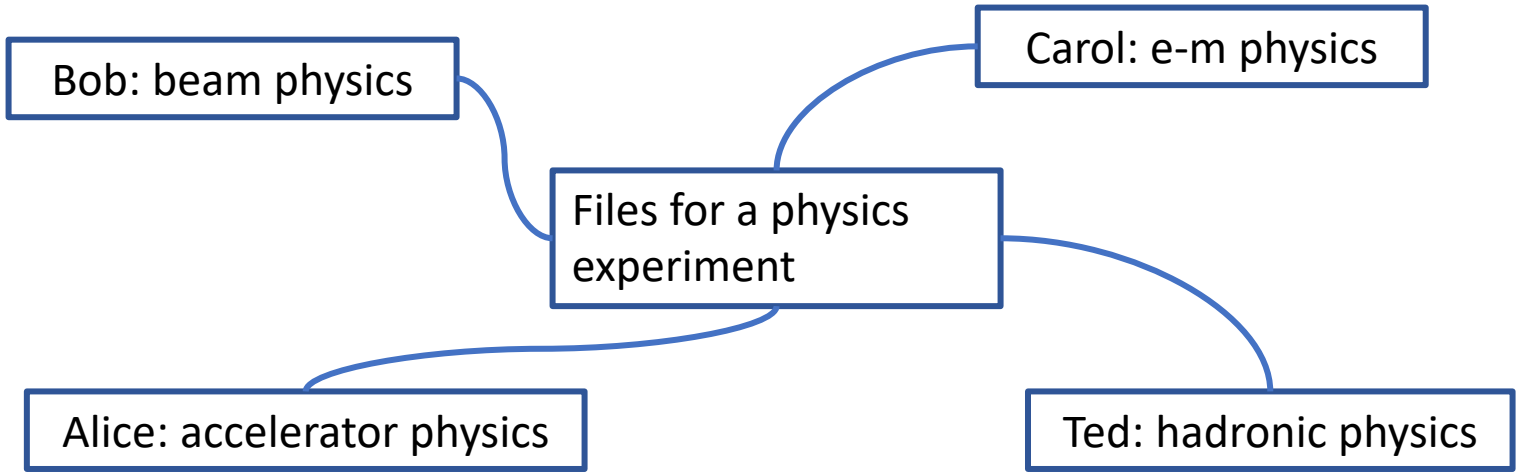

```
git checkout main # back to the 'main' branch to do some other work
emacs myfile.py
git commit -a -m "a main branch revision"

git checkout parsing # switch back and forth between branches as needed
emacs whatever.txt
git commit -a -m "more 'parsing' changes"

git checkout main # To go back to the main branch before the merge
git merge parsing # To incorporate the 'parsing' changes into your main files
```

See the "git" section in the ROOT tutorial's appendices for details.

Where? Perhaps a git repository

A "git repository" is a central location for a project's files. Most physics projects are multi-user, with several people working on their files at once (a programming project, a science paper, etc.).

While a git repository can be on almost any computer, most of the projects you'll see among the Nevis working groups use GitHub. *(Insert editorial here.)*



In the example above, a group of physicists are all working from the same code base. Perhaps they're each creating their own branches as they work on their areas of expertise. At some point they'll coordinate with each other (larger experiments have a software manager) to merge their work into the code in the central repository.

Accessing a git repository

Your experimental group will let you know how they'd like you to work with git (if they work with it at all):

- Git is already installed on every system on the Nevis Linux particle-physics cluster.
- If they're using a common service (e.g., codeberg), you'll have to create an account there if you don't already have one.
- They may ask you to save your work as an independent unit:
  - Create a repository on the service. Assume your service username is **`jsmith`** and the repository's name is **`NevisWork`**.
  - Within your git-managed directory on your local system (assuming your main branch is **`main`**):

```
git remote add origin https://codeberg.org/jsmith/NevisWork.git
git push origin main
```

- If your group has an existing repository, they'll give you instructions on how to download it to a local directory. It will probably look something like this:

```
git clone git@codeberg.org:OWNER/REPOSITORY.git
cd REPOSITORY
git fetch
```

How does git track changes between commits?

The details are involved, but basically it uses the UNIX **diff** command:

test.txt

```
This will create a directory in your
area whose name is “REPOSITORY”. You
do not create that directory in advance;
git clone will create it for you.

Note that the text OWNER and REPOSITORY
in the above command are examples. Your
working group has to tell you the GitHub
name of the owner of the repository,
and the name of the repository.
```

test2.txt

```
This will create a directory in your
area whose name is “REPOSITORY”. You
don't create that directory in advance;
git clone will create it for you.

Note that the text "OWNER" and "REPOSITORY"
in the above command are examples. Your
working group has to tell you the GitHub
name of the owner of the repository,
and the name of the repository.
```

The result of using diff:

```
$ diff test.txt test2.txt
3c3
< do not create that directory in advance;
---
> don't create that directory in advance;
6c6
< Note that the text OWNER and REPOSITORY
---
> Note that the text "OWNER" and "REPOSITORY"
```

diff returns a set of instructions of how to go from one file to the other.

This matters because…

- **diff** is good for determining the differences as you edit text files; e.g.,:
  - program code
  - shell scripts
  - options files
  - research papers
- Git stores the differences between commits in a "hidden directory" `.git`
- But **diff** can't produce meaningful results when comparing binary files; e.g.,:
  - compiled programs and libraries
  - most databases
  - ROOT n-tuples
  - images

➢ As a general rule, do not execute `git add <file>` if the file is a binary file. ("`git add .`" is usually a bad idea.)

➢ Exception: The file is completely static (e.g., the .png files I use in the ROOT tutorial).

When do we use tags?

Short answer: When your group tells you to do so.

- A tag is a "bookmark" assigned to a commit by a human.
  - Compare this to an ID like `f19e6e6` created by git (shown in `git reflog`)
  - A tag is typically used to assign a version number to software as it evolves.
- To see a list of tags in your repository:

```
git tag -n1
```

- To create a tag (v3.141 in this example):

```
git tag -a v3.141 -m "A comment on why this tag was added"
```

- Tags are *not* automatically transferred with a generic `git push` or `git pull`
  - You have to individually push them:

```
git push v3.141
```

  - You can pull any and all tags (and all branches) in the remote repository with

```
git fetch
```

- If you want to revert your files to a specific tag:

```
git checkout v3.141
```

The End
(for real this time!)