Introduction to GIT - Version Control

A **version control system** is a tool that keeps track of changes in a folder or folders and all the files there in.

This allow us to store our "package" in a single place, and being able to record how it changes over time. This will happen in the repository. Even if you make a mistake, you can always go back to a previous version. One of the cool parts is that many people can work on the same project or you can work from different machines.

The approach is simple, it uses a distributed approach, which means that EACH one of the users have a copy of the project and also has a copy of the history of the changes. As it is the most used VCS, it is now very important that you know how it works. We will use to received and hand in our homeworks!

You can use GIT

1. In the command line (the fastest and the easiest)
2. Most code editors or IDE have already an interface with GIT... we will not use it here but at least you know they exist
3. A graphical unit interface. Depending on your system (Windows, linux, etc) you need to use a different package. If interested, check https://git-scm.com/downloads/guis. (Check GitKraken or SourceTree, the last only in Windows and Mac).

Here **we will use the command line**, as it is more powerful and it is always "simple to use".

The first step is to installe git.... lucky we have done so. To check

```
git --version
```

If you want to installed in your Windows device (something I do not know how to do), after installing git, you need to download the program "git bash" because it provides a BASH emulation used to run GIT from the command line.

The first time we use Git, we need to configure it. These settings correspond to name, email, default editor (default in Git is vi), line ending. We can always do it at the system level (if you are superuser) and it is applied to all users, at the global level, which applies to all repositories of the current level and then local where the settings only apply to the directory you are.

```
git config --global user.name "aldo romero"
git config --global user.email alromero@mail.wvu.edu
git config --global core.editor vi

# now let set I want to edit the info which is in this file
git config --global -e

In my case, I will see a file that contains
```

```
[user]
        name = aldo romero
        email = alromero@mail.wvu.edu
[filter "hawser"]
        clean = git hawser clean %f
        smudge = git hawser smudge %f
        required = true
[core]
        editor = vi
```

Now, to configure end of lines, we need to remember that end of line depends on the system, in windows you have: aldo\r\n (the \r is the carriage return and \n is the line feed), in Mac or Linux is just aldo\n

Now, to address issues with the end of line, we need to configure a property called "core.autocrlf", this is done by

```
git config --global core.autocrlf true
        (true if you are in windows or input if you are in Mac or Linux)
```

If you want to get a full list of commands for "git config", you can google it or you can

```
git config --help

or a short summary is

git config -h
```

For example, you can create a directory, let me called Romero_Computational_Physics

and within, I will store every one of the homeworks. The idea for this case, it is that

1. You can make a copy of what is in your hard drive into some place in the cloud. It will keep the structure.
2. You can allow people to access this directory, for example, me if I want to grade the HW.
3. You can have different version of the homework, save all of them and only submit the one you are happy with.
4. You decide which changes will be made to the next version (each record of these changes is called a commit), and keeps useful metadata about them (like, when, how, what changes, etc).
5. The complete history of commits for a particular project and their metadata make up a repository.
6. Repositories can be kept in sync across different computers, facilitating collaboration among different people.

In scientific computing there are 3 typical scenarios where using a Version Control system is crucial in your effectiveness as a researcher.

1. You write code, scripts, and sometimes even data, and you need to keep track of the changes on that code in case you need to go back and review why a previous version was producing better results!.
2. You are writing a paper: Your paper will go into a spiral of reviews by you, your collaborators, your Ph.D. advisor, the referees. Suppose you are using LateX for your paper, as you probably should do in the future. In that case, it is easy to keep track of changes and keep your paper in sync across several computers.
3. You are writing your Thesis, the same principle; your manuscript will have several cycles of review and corrections. You do not want to keep all your precious Ph.D. Thesis on a single place, and you do not want to get lost figuring out which version is the last!.

Version control systems can help you with this, they are designed to keep track of changes on files. Even if they work also for binary files like figures and plots, adding too many figures, photos or other media could make the repository grow quickly.

For the purpose of this hands-on, we will create and work entirely on an local repository. You can otherwise create an account on Github (GitHub.com) and get the ability to create public repositories for free or private repositories for a fee. WVU Research Computing offers to researchers private repositories for free via an special agreement with Github.

Exercise 1 (Create an empty repository)

In this exercise you will create an empty repository from the command line. Before that, there are a few commands to prepare your environment for using with Github:

```
git config --global user.name "<Your Full Name>"
git config --global user.email "<username>@mix.wvu.edu"
```

The default editor for commits is `vim` and for the purpose of this introduction we will keep it like that.

Now, lets create a folder `project_01` that we will convert into a git repository so Git can store versions of our files. Move inside the folder (`cd project_01`) and execute:

```
$ git init
```

after this, you will see a message (in my machine):

```
Initialized empty Git repository in
/Users/aldoromero/Class/ComputationalPhysics/MyClass/GitHub_Intro/.git/
```

If we use `ls` to show the directory's contents you will see no new file apparently.

```
$ ls
```

But if we add the `-a` flag to show hidden files and folders, we can see that Git has created a hidden folder within `project_01` called `.git`:

```
$ ls -a
. ..  .git
```

Git uses this special sub-directory to store all the information about the project, including all files and sub-directories located within the project's directory. If we ever delete the `.git` sub-directory, we will lose the project's history. That folder is hidden for a good reason, in almost all circumstances, you are not suppose to manipulate the contents of the `.git` folder directly.

We can check that everything is set up correctly by asking Git to tell us the status of our project:

```
$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
```

We have now a repository, an empty one, lets start by adding some basic structure to it. Lets suppose that the project will include: the scripts that you use to create or manipulate the data, a curated set of data, the plots and the manuscript of your paper.

It is not a good idea to store the raw data, raw data is probably too big, we just need the data that will be actually used to create the plots.

A git repository is not indented to be a file backup. What we should keep in the repository is the code and text that allows to recreate the results from the raw data, if the raw data is lost (something that should never happen if you backup your data), you can reconstruct the results, by running all simulations again, using your scripts, producing equivalent plots and recreating your paper for submission.

Exercise 2 (Adding files to the repository)

Lets create the structure for `project_01`

```
mkdir data
mkdir scripts
mkdir plots
mkdir paper
touch data/README.md
touch scripts/README.md
touch plots/README.md
touch paper/README.md
```

With `touch` we are creating some empty files, git does not allow us to add empty folders. Those README.md should contain instructions about the data collected on those folders, how the scripts and plots should operate, of how the paper should be built. Research is a lengthy endeavor; do not assume you will remember what you did several months ago. Now we are ready to version control those folders.

```
$ git add data paper plots scripts
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

 new file:   data/README.md
 new file:   paper/README.md
 new file:   plots/README.md
 new file:   scripts/README.md
```

Now is time for our first commit:

```
$ git commit -m "My first commit"
[master (root-commit) 5b5a7ed] My first commit
 4 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 data/README.md
 create mode 100644 paper/README.md
 create mode 100644 plots/README.md
 create mode 100644 scripts/README.md
```

When we run `git commit`, Git takes everything we have told it to save by using git add (right now empty files) and stores a copy permanently inside the special .git directory. This permanent copy is called a commit (or revision) and its short identifier is 5b5a7ed (or any string you get in your case). Each identifier is different so you will see a different code.

Ok, this has been helpful but let us go un step back. Let me start with the commands we will cover now in more detail (well no all but at least we will have them accessible for most of the possibilities)

```
#initialize a repository
git init

#staging files
git add aldo.tex   # Staging a single file
git add aldo.tex aldo1.tex aldo2.tex
git add *tex       # Staging with a pattern
git add .          # Staging the current directory and all its content

#viewing the status
git status         # full status
git status -s      # short status

#commiting the staged files
git commit -m "message"

#Skipping the staging area
git commit -am “Message”

#Removing files
git rm file1.js     # remove from working directory and staging area
git rm --cached file1.js # remove from staging area only

#Renaming or moving files
git mv file1.js file1.txt

#Viewing the staged/unstaged changes
git diff              # Shows unstaged changes
git diff --staged     # Shows staged changes
git diff --cached     # Same as the above

#Viewing the history
git log               # full history
git log --oneline     # summary
git log --reverse     # list the commits from the oldest to the newest

#Viewing a commit
git show 921a2ff      # show the given commit
git show HEAD         # shows the last commit
git show HEAD~2       # Two steps before the last commit
git show HEAD:aldo.txt   # hows the version of the file aldo.txt from the last commit

#Unstaging files (undoing git add)
```

```
git restore --staged aldo.txt   # Copies the last version of aldo.txt from repo to index

#Discarding local changes
git restore aldo.txt   # Copies aldo.txt from index to working directory
git restore aldo1.txt aldo2.txt   # Restores multiple files in working directory
git restore .   # Discards all local changes (except untracked files)
git clean -fd     # Removes all untracked files

#Restoring an earlier version of a file
git restore --source=HEAD~2 aldo.txt

# Viewing the history
git log --stat          # Shows the list of modified files
git log --patch         # Shows the actual changes (patches)

#Filtering the history
git log -3              # Shows the last 3 entries
git log --author="Aldo"
git log --before="2020-08-17"
git log --after="one week ago"
git log --grep="GUI"    # Commits with "GUI" in their message
git log -S"GUI"         # Commits with "GUI" in their patches
git log hash1..hash2    # Range of commits
git log file.txt        # Commits that touched file.txt

#Formatting the log output
git log --pretty=format:"%an committed %H"

#Creating an alias
git config --global alias.lg "log --oneline"

#Viewing a commit
git show HEAD~2
git show HEAD~2:file1.txt # Shows the version of file stored in this commit

#Comparing commits
git diff HEAD~2 HEAD # Shows the changes between two commits
git diff HEAD~2 HEAD file.txt # Changes to file.txt only

#Checking out a commit
git checkout dad47ed     # Checks out the given commit
git checkout master      # Checks out the master branch

#Finding a bad commit
git bisect start
git bisect bad           # Marks the current commit as a bad commit
```

```
git bisect good ca49180  # Marks the given commit as a good commit
git bisect reset         # Terminates the bisect session

#Finding contributors
git shortlog

#Viewing the history of a file
git log file.txt        # Shows the commits that touched file.txt
git log --stat file.txt   # Shows statistics (the number of changes) for file.txt
git log --patch file.txt   # Shows the patches (changes) applied to file.txt

#Finding the author of lines
git blame file.txt               # Shows the author of each line in file.txt

#Tagging
git tag v1.0                     # Tags the last commit as v1.0
git tag v1.0 5e7a828             # Tags an earlier commit
git tag                          # Lists all the tags
git tag -d v1.0                  # Deletes the given tag

#Managing branches
git branch fixedbug      # Creates a new branch called fixedbug
git checkout fixedbug    # Switches to the fixedbug branch
git switch fixedbug      # Same as the above
git switch -C fixedbug   # Creates and switches
git branch -d fixedbug   # Deletes the bugfix branch

#Comparing branches
git log master..fixedbug    # Lists the commits in the bugfix branch not in master
git diff master..fixedbug   # Shows the summary of changes

#Stashing
git stash push -m “New tax rules”      # Creates a new stash
git stash list                         # Lists all the stashes
git stash show stash@{1}               # Shows the given stash
git stash show 1                       # shortcut for stash@{1}
git stash apply 1                      # Applies the given stash to the working dir
git stash drop 1                       # Deletes the given stash
git stash clear                        # Deletes all the stashes

#Merging
git merge fixedbug                      # Merges the bugfix branch into the current branch
git merge --no-ff fixedbug              # Creates a merge commit even if FF is possible
git merge --squash fixedbug             # Performs a squash merge
git merge --abort                      # Aborts the merge
```

```
#Viewing the merged branches
git branch --merged                     # Shows the merged branches
git branch --no-merged                  # Shows the unmerged branches

#Rebasing
git rebase master                       # Changes the base of the current branch

#Cherry picking
git cherry-pick dad47ed                 # Applies the given commit on the current branch
```

Creating Snapshots of your project

First we need to start a project, this is by creating a directory

```
mkdir CP_HW1
cd CP_HW1
# now we need to initialize what is in this directory as a git repository
git init
# to see what is in this directory use  ls -la

Now this is what you can do to see what is in this directory

cd .git
ls

The output of this command is

HEAD  config  description  hooks  info  objects  refs

Now, remember, WE DO NOT TOUCH THIS DIRECTORY. It has been created and it will be
maintained by Git
```

Ok, now we know how to initialize the project, let see the usual workflow of a project.

First, let say we modify or create or do something into files in this directory. After we have done all changes, we need to commit all these changes into the repository.

This makes use of a very specific propery of GIT.**It has an intermediate area between your development and the repository**. This area is call the **staging area** or the index, which stores the changes or modifications we have done for our next commit. Therefore, before we commit, we need to add all the files to the staging area or index, review that everything is good and then we do the commit.

Let me work an example

```
#Right now, our current directory is empty
```

```
aldoromero@x86_64-apple-darwin13 CP_HW1 % ls
aldoromero@x86_64-apple-darwin13 CP_HW1 %

#let me create two files

aldoromero@x86_64-apple-darwin13 CP_HW1 % touch file1.tex
aldoromero@x86_64-apple-darwin13 CP_HW1 % touch file2.tex
aldoromero@x86_64-apple-darwin13 CP_HW1 % ls
file1.tex file2.tex

# we can check what is going on with the project before we put info in the staging area
git status

#now we add the files into the staging area
aldoromero@x86_64-apple-darwin13 CP_HW1 % git add file1.tex file2.tex
aldoromero@x86_64-apple-darwin13 CP_HW1 %

#if everything is good, now we commit these files into the repository
aldoromero@x86_64-apple-darwin13 CP_HW1 % git commit -m "initial commit"
[master (root-commit) 1e62045] initial commit
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.tex
 create mode 100644 file2.tex

 # even we have commmited, the staged area is not empty.

 # now let us make changes to one if the files
 aldoromero@x86_64-apple-darwin13 CP_HW1 % vi file1.tex

 # after this change, in staging area we have the old version of file1.tex
 # to let the staging area to know that we have changed the file

aldoromero@x86_64-apple-darwin13 CP_HW1 % git add file1.tex
aldoromero@x86_64-apple-darwin13 CP_HW1 %

# Now, in staging area we have the new version of the file
aldoromero@x86_64-apple-darwin13 CP_HW1 % git commit -m "fixing a problem"
[master 41fd8ff] fixing a problem
 1 file changed, 1 insertion(+)

#now let us decide that we do not need file2.tex
# we delete it from our working directory

aldoromero@x86_64-apple-darwin13 CP_HW1 % rm file2.tex
aldoromero@x86_64-apple-darwin13 CP_HW1 % git add file2.tex
```

```
# now the staging area knows that file2 has been deleted, though we used "add"
# now we commit again

aldoromero@x86_64-apple-darwin13 CP_HW1 % git commit -m "deleting file2"
[master 7c13fed] deleting file2
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 file2.tex

#now, each commit has a unique identifier that was created in our repository

#let me explain a bit more by creating again file2.tex
aldoromero@x86_64-apple-darwin13 CP_HW1 % echo newthing > file2.tex
aldoromero@x86_64-apple-darwin13 CP_HW1 % git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
  file2.tex

nothing added to commit but untracked files present (use "git add" to track)

aldoromero@x86_64-apple-darwin13 CP_HW1 % git add file2.tex
aldoromero@x86_64-apple-darwin13 CP_HW1 % git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  new file:   file2.tex

aldoromero@x86_64-apple-darwin13 CP_HW1 % echo newthing >> file2.tex
aldoromero@x86_64-apple-darwin13 CP_HW1 % git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  new file:   file2.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
  modified:   file2.tex

#Now you see that in the staginh area we have file2.tex and even without commit we also
have file2.tex. Both are different!

#if you want to be more specific about your commit, you can use
git commit
#this line will open your editor and you can explain in long words what you want to say
```

```
#BUT this has to be done in two steps, when your editor open, you enter a short
# description in the first line then leave a line in blank and then in the following you
can write a long description.

aldoromero@x86_64-apple-darwin13 CP_HW1 % git commit -m "including file2.tex again"
[master b59a394] including file2.tex again
 1 file changed, 1 insertion(+)
 create mode 100644 file2.tex
```

Now, before we go on, I would like to stress something on the commit. Please avoid to commit with very tiny and small changes. Otherwise, git will become bigger and bigger.. just commit often! which logical separations between different commits

Skipping stiging area

Let say you have change a file in your working directory and you want to commit this without the "add" step.

```
git commit -am "new file to be stored"
```

Well, it is useful but it is not recommended at the beginning.. it is TOO dangerous.

Removing files

Let say we want to remove a file from the repository as we do not need it anymore.

```
$ git status
On branch master
nothing to commit, working tree clean

#now we see the files in the staging area
$ git ls-files
file1.tex
file2.tex
$ rm file2.tex
$ git ls-files
file1.tex
file2.tex
$ ls
file1.tex
# here we can see that we delete the file but the siting area does not know it.
# to make the change to happen, let see

$ git add file2.tex
$ git ls-files
file1.tex
$ git status
```

```
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  deleted:    file2.tex

$ git commit -m "remove unused file"
[master 5be2539] remove unused file
 1 file changed, 3 deletions(-)
 delete mode 100644 file2.tex

 #As this is very common, we can use a Git command to do this at once.
 # delete the file from the staging area and the repositoty at once

 git rm file2.tex
```

Moving files

```
$ ls
file1.tex file2.tex
$ git mv file2.tex file3.tex
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
  renamed:    file2.tex -> file3.tex
$git commit -m "using mv to rename files"
```

Ignoring Files

For example exe or executable files, etc. The following method works for the first time.. if you already pass the files to the stiging area, then, you can not do this.

```
$ mkdir StoredLogs
$ echo logs > StoredLogs/n.log
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
  StoredLogs/

nothing added to commit but untracked files present (use "git add" to track)
```

```
# create a file .gitignore with the list of files to ignore
$ echo StoredLogs > .gitignore
$ .git status
zsh: command not found: .git
aldoromero@x86_64-apple-darwin13 CP_HW1 % git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
  .gitignore

nothing added to commit but untracked files present (use "git add" to track)

# now you see that .gitignore is not commited by the directory dissapear

$ git add .gitignore
```

Therefore, if you commit a file in the staging area by mistake, then, everytime you commit, this file will be tracked and you do not want that, you need to delete this file from the staging area. To do so

```
#remove fromt the index
git rm --cached -r StoredLogs
```

As the files you will be ignoring are "language dependent", there are some recommendations from GIT developers. Take a look to

GitHub.com/github/gitignore/blob/master

Short Status

```
$ echo nothingelse1 > file3.tex
$ echo nothingelse1 >> file3.tex
$ cat file3.tex
nothingelse1
nothingelse1
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
  modified:   file3.tex

no changes added to commit (use "git add" and/or "git commit -a")
$ echo nothingelse1 >> file2.tex
$ git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
  modified:   file3.tex

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  file2.tex

no changes added to commit (use "git add" and/or "git commit -a")
$ git status -s
 M file3.tex
?? file2.tex
$ git add file3.tex
# it changes color
$ git status -s
M  file3.tex
?? file2.tex
$ echo nothingelse1 >> file3.tex
$ git status -s
MM file3.tex
?? file2.tex
$ git add file3.tex
$ git status -s
M  file3.tex
?? file2.tex
$ git add file2.tex
$ git status -s
A  file2.tex
M  file3.tex

# NOW BEFORE YOU COMMIT< ALWAYS CHECK WHAT YOU HAVE IN THE STAGING AREA
# The following command will compare copies of the same file
# to read it... one of the lines is important
#
#  @@ -0,0 +1 @@
#      -0,0 means that from the old file   -0 no changes , 0 no show any line
#       but we did not have file2.tex, therefore there is nothing to compare
#       now it is the new file
#      +1  it will show one line starting from the first line


$ git diff --staged
diff --git a/file2.tex b/file2.tex
new file mode 100644
```

```
index 0000000..3affe23
--- /dev/null
+++ b/file2.tex
@@ -0,0 +1 @@
+nothingelse1
diff --git a/file3.tex b/file3.tex
index 9dafe9b..6641a77 100644
--- a/file3.tex
+++ b/file3.tex
@@ -1 +1,3 @@
-nothing
+nothingelse1
+nothingelse1
+nothingelse1

#now let see if the differences are comming in our actual directory
$git diff
$

#Nothing is reported, meaning that all changes are already in the staging area

# Now change file2.tex in our working area and see differences between
# working area and staging area

$ echo zero > file2.tex
$ git diff
diff --git a/file2.tex b/file2.tex
index 3affe23..26af6a8 100644
--- a/file2.tex
+++ b/file2.tex
@@ -1 +1 @@
-nothingelse1
+zero
```

Viewing the History

Use the following command to see the history

```
git log
```

For example, in my case

```
commit b1f2feace97811bf0707bdb606526d8d3e8f8c7d (HEAD -> master)
Author: aldo romero <alromero@mail.wvu.edu>
Date:   Thu Aug 19 17:32:26 2021 -0400

    recent changes

commit ac043232ab64053f67daf4e31223fa5bb3bea031
Author: aldo romero <alromero@mail.wvu.edu>
Date:   Thu Aug 19 17:16:05 2021 -0400

    using mv to rename files

etc.....
```

```
$ git log --oneline
b1f2fea (HEAD -> master) recent changes
ac04323 using mv to rename files
92f58a2 fixing a problem
1595685 moving file2 into file3
7597c1f adding file again
5be2539 remove unused file
96d20b5 new new file added
3100d7d3 new file added
b59a394 including file2.tex again
7c13fed deleting file2
41fd8ff fixing a problem
1e62045 initial commit


# if you want to reverse the order
$ git log --oneline --reverse
```

Viewing a commit

See the changes in a given commit

```
# we only use any number as long as it is different than the others
git show 92f5
#To look the last commit
git show HEAD
#The previous one
git show HEAD~1

For example if we use the last command
```

```
$ git show HEAD~1
commit ac043232ab64053f67daf4e31223fa5bb3bea031
Author: aldo romero <alromero@mail.wvu.edu>
Date:   Thu Aug 19 17:16:05 2021 -0400

    using mv to rename files

diff --git a/file2.tex b/file3.tex
similarity index 100%
rename from file2.tex
rename to file3.tex


If we do not want the differences but the final version of that commit

$ git show HEAD:.gitignore
StoredLogs

If you want to see all files and directories in a commit, in this notation, files are blob
and directories should be tree

$ git ls-tree HEAD~1
100644 blob 8e601be92aaf38c6cdb45bb4b2643dd05cd0c060  file1.tex
100644 blob 9dafe9be2099a3b02b618c673bcf865a1ffb4f9d  file3.tex

If you want to see the content

$ git show 8e601be92aaf38c6cdb45bb4b2643dd05cd0c060
This is new
```

Unstaging Files

You always have to review the staging area before you commit. You can delete files from the stating as you do not want to commit. It is ti "undo" the add. This will use the command

```
git restore --staged file3.tex     (you can use . as all files)
```

What this is doing, it is copying from the repository to the staging area.

Discard Local Changes

```
git restore .
```

It could happen that you can have files that you created but that Git is not tracking. Therefore, we need to delete it, this can be done if you do

```
git clean
```

git will complain as if you allow this, it will delete everything, you can check the possibilities

```
git clean -h
```

then, after reading it, you can find that the best would be

```
git clean -fd
```

Restoring a file from a previous change

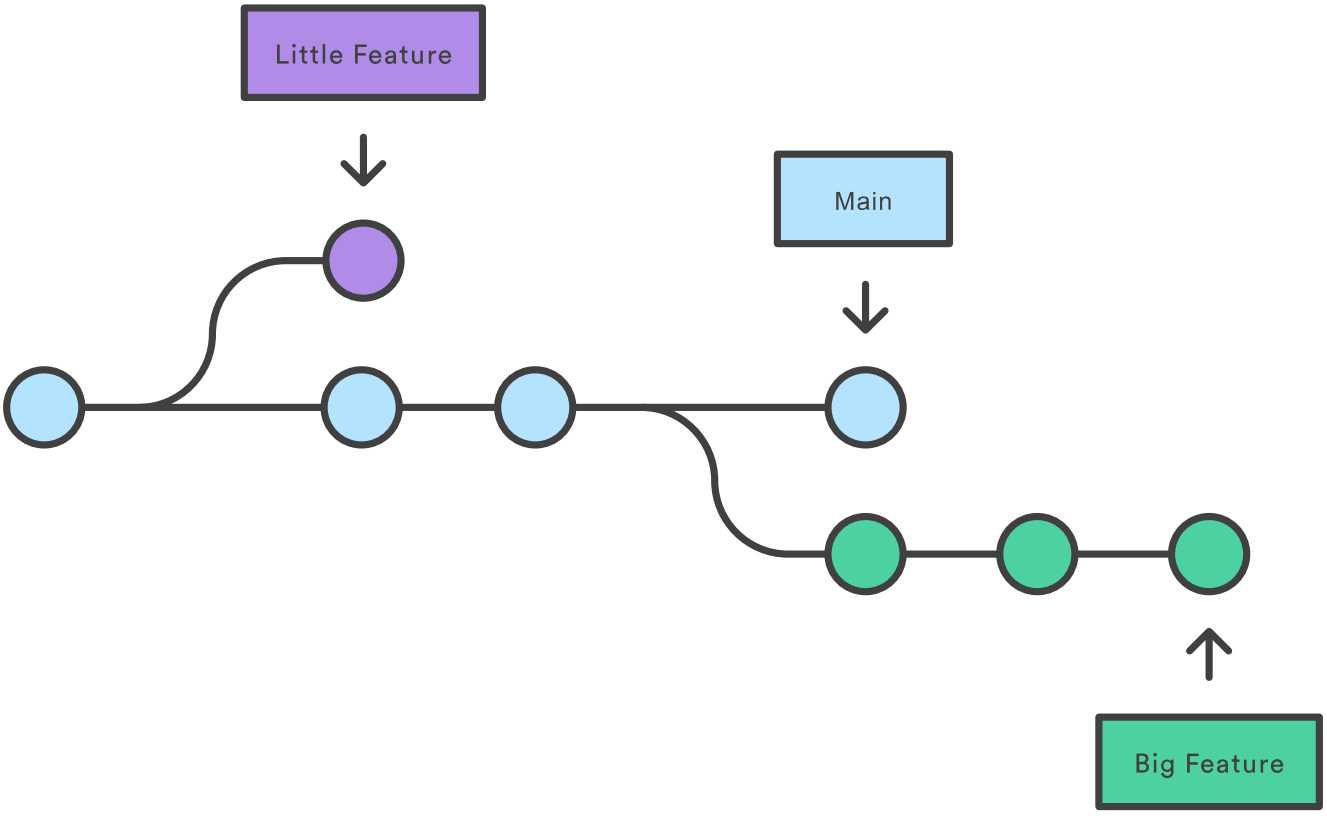
```
#let us delete a file from the directory and the staging area
$ git rm file2.tex
rm 'file2.tex'


$ git status -s
D  file2.tex

#and we commit
$ git commit -m "shout I delete file 2"
[master 6dcae26] shout I delete file 2
 1 file changed, 1 deletion(-)
 delete mode 100644 file2.tex

#now let us recover that file

$ git log --oneline
6dcae26 (HEAD -> master) shout I delete file 2
22e0ce4 new mod
f9d6467 last changes 1
ca8d09a last changes
b1f2fea recent changes
ac04323 using mv to rename files
92f58a2 fixing a problem
```

```
1595685 moving file2 into file3
7597c1f adding file again
5be2539 remove unused file
96d20b5 new new file added
310d7d3 new file added
b59a394 including file2.tex again
7c13fed deleting file2
41fd8ff fixing a problem
1e62045 initial commit

$ git restore -h
usage: git restore [<options>] [--source=<branch>] <file>...

    -s, --source <tree-ish>
                          which tree-ish to checkout from
    -S, --staged          restore the index
    -W, --worktree        restore the working tree (default)
    --ignore-unmerged     ignore unmerged entries
    --overlay             use overlay mode
    -q, --quiet           suppress progress reporting
    --recurse-submodules[=<checkout>]
                          control recursive updating of submodules
    --progress            force progress reporting
    -m, --merge           perform a 3-way merge with the new branch
    --conflict <style>    conflict style (merge or diff3)
    -2, --ours            checkout our version for unmerged files
    -3, --theirs          checkout their version for unmerged files
    -p, --patch           select hunks interactively
    --ignore-skip-worktree-bits
                          do not limit pathspecs to sparse entries only
    --pathspec-from-file <file>
                          read pathspec from file
    --pathspec-file-nul   with --pathspec-from-file, pathspec elements are separated with
NUL character

$ git restore --source=HEAD~1 file2.tex
$ git status -s
?? file2.tex
```

Ok, now we manage git in your directory and you can work your project, now let us connect our work with GitHub.

Branches

Branching in other VCS's can be an expensive operation in both time and disk space. In Git, branches are a part of your everyday development process. Git branches are effectively a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes.



A branch represents an independent line of development.

The `git branch` command lets you create, list, rename, and delete branches. It doesn't let you switch between branches or put a forked history back together again. For this reason, `git branch` is tightly integrated with the `git checkout` and `git merge` commands.

```
git branch

# Create a new branch called <branch>

git branch <branch>

# Delete the specified branch. This is a "safe" operation in that Git prevents you
# from deleting the branch if it has unmerged changes.
# This only works if the branch has been commited

git branch -d <branch>
```

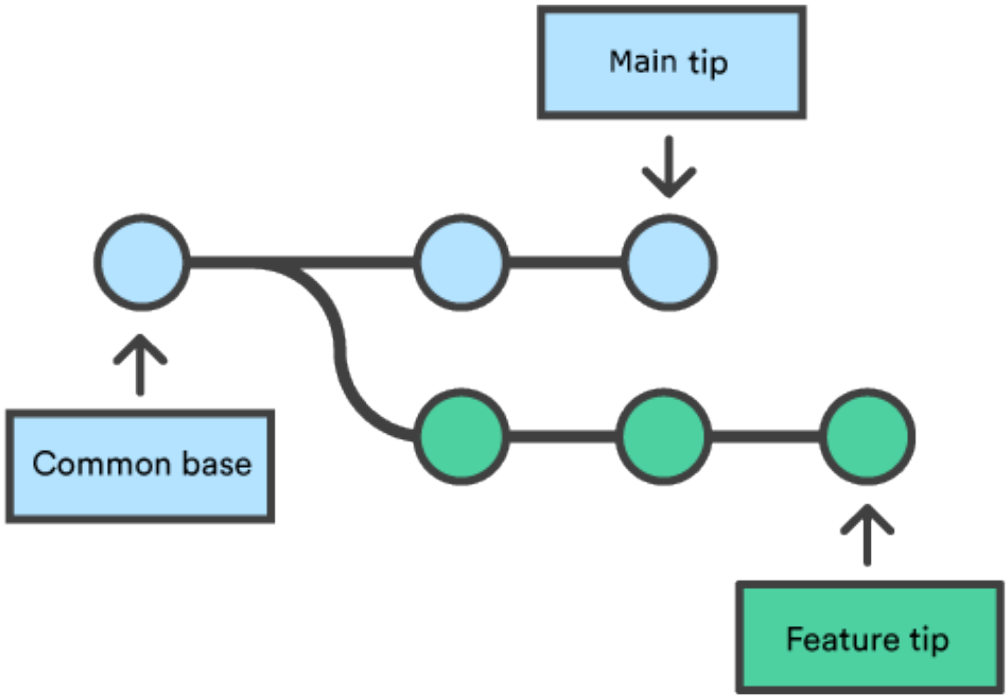
```
# Force delete the specified branch, even if it has unmerged changes.
# This is the command to use if you want to permanently throw away all of the commits
# associated with a particular line of development.

git branch -D <branch>

# Rename the current branch to <branch>.

git branch -m <branch>
```

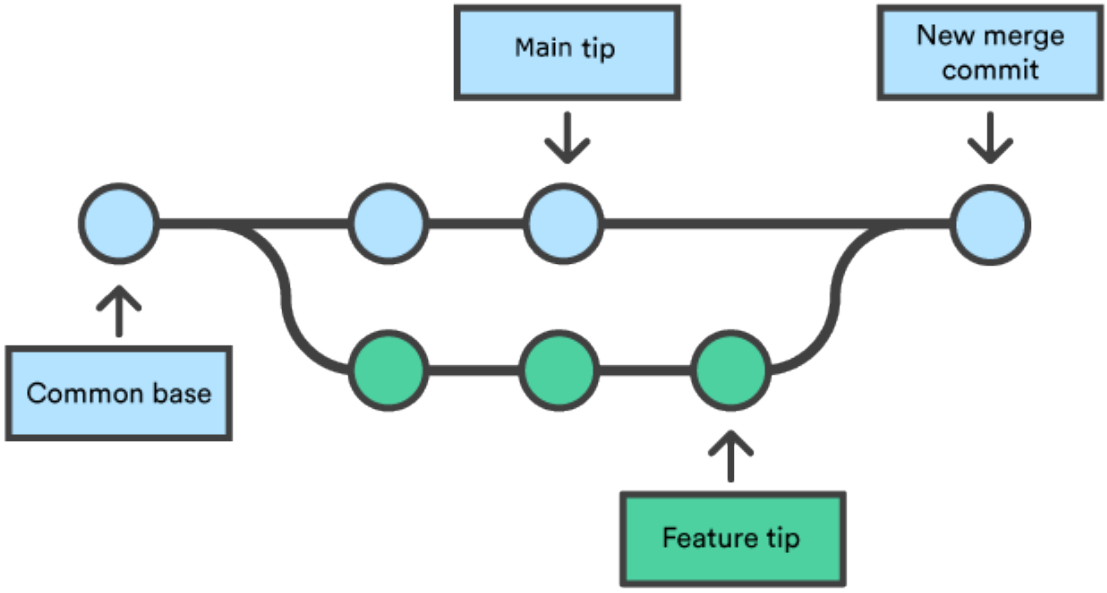
To start adding commits to the corresponding branch, you need to select it with `git checkout`, and then use the standard `git add` and `git commit` commands. In Git terms, a "checkout" is the act of switching between different versions of a target entity

The `git checkout` command lets you navigate between the branches created by `git branch`.

```
# For example

$> git branch
main
another_branch
this-is-another-branch

$> git checkout this-is-another-branch

# The git checkout command accepts a -b argument that acts as a convenience method
# which will create the new branch and immediately switch to it.

git checkout -b <new-branch>

# IMPORTANT: By default git checkout -b will base the new-branch off the current HEAD
# if you want to use a different branch

git checkout -b <new-branch> <existing-branch>

# To switch between branches

git checkout <branchname>
```

Now let say, you are done with your changes and you want to merge them. Merging is Git's way of putting a forked history back together again. `Git merge` will combine multiple sequences of commits into one unified history. Like this:





Before performing a merge there are a couple of preparation steps to take to ensure the merge goes smoothly.

1. Execute `git status` to ensure that `HEAD` is pointing to the correct merge-receiving branch. If needed, execute `git checkout` to switch to the receiving branch. In our case we will execute `git checkout main.`
2. Make sure the receiving branch and the merging branch are up-to-date with the latest remote changes.

> Execute `git fetch` to pull the latest remote commits. Once the fetch is completed ensure the `main` branch has the latest updates by executing `git pull.`

Ok, I see.. you are not familiar with teh git fetch.

`git pull` and `git fetch` commands are available to download content from a remote repo. You can consider `git fetch` the 'safe' version of the two commands. It will download the remote content but not update your local repo's working state, leaving your current work intact. `git pull` is the more aggressive alternative; it will download the remote content for the active local branch and immediately execute `git merge` to create a merge commit for the new remote content.

Once the previously discussed "preparing to merge" steps have been taken a merge can be initiated by executing `git merge` where `` is the name of the branch that will be merged into the receiving branch.

```
# Start a new feature
git checkout -b new-feature main
# Edit some files
git add <file>
git commit -m "Start a feature"
# Edit some files
git add <file>
git commit -m "Finish a feature"
# Merge in the new-feature branch
git checkout main
git merge new-feature
git branch -d new-feature
```

Ok... this is becoming complicated... but I hope you got the main idea....

Creating a GitHub Repository

You need

1. Know Git (basic)
2. Git in your system
3. A GitHub account

Now that you have created your repository and you have made changes and everything else, now you need to push this to GitHub

In your browser, go to github.com and log in if you need to. Click the plus sign icon at the top right of the page. Then select **New Repository**.

Enter a name, then click **Create Repository**. I have used the name `Repository-CP301-WVU-Romero` for this tutorial. We will be using this name later on, so make a note of it.

There you have it, a sparkling new repository. If this is your first one, congratulations!!!

You have reached a programming milestone.

Stay on the **Create Repository** page to complete the next step.

So far in our project folder we have just one or some file(s), and we have committed the changes we made to it. The next step is to push these changes to GitHub. Our project is in 'Repository-CP301-WVU-Romero', which we just created, so we can push to an existing repository.

Click the `clipboard` icon to copy the three commands shown in the "push an existing repository from the command line' section. Those lines should contains things like

```
git remote add origin ....
git branch -M ..
git push -u ....
```

Paste them in your Terminal and press **ENTER** to execute them but before ***Replace the username shown in the code with your GitHub username*.**

At the end, it will be something very similar to

```
git remote add origin https://github.com/AldoRomero/Repository-CP301-WVU-Romero.git
git branch -M main
git push -u origin main
```

After running these commands, reload the browser page. Our `files or the index.html` file is now listed in the online repository.

You can make more updates to the repository by running these commands in order:

```
git add .
git commit -m "Commit message"
git push origin main
```

If you want to make a copy into a different machine

```
git clone <gitaddress>
```

If you want to see changes

```
git push
```

Now, use google to find anything that I have missed :-)