**APPM 4600 Lab 2**
Set up your Git repository

## 1 Overview

In this lab, you will learn about *version control* software and how to manage your toolbox with it. Version control software is useful for tracking the development of a code, as well as other large projects that consist of text files, like your final project. You will manage your toolbox with a repository on the platform Github, at `https://github.com/`.

**Definition 1.1.** A *repository* or "repo" is the set of code and its development history, stored in some platform.

When using git, there are actually two versions of the repository: one on your machine and one online. If you share the repository with others, there is a copy on each person's machine. Your goal through out the process is to make changes locally and upload them to the online repository. To keep the online repository current, it is best to update your online repo regularly; i.e. after any edits to the local repository and if taking a break. Section 3.8 will review some good git habits and point you in the direction of more resources.

## 2 Before lab

You will be accessing your repo via `ssh`[1]. Before the lab, you should do the following:

1. Go to https://github.com/ and sign up for an account.
2. Make sure that you know how to access and use your terminal.
3. Install Git on your local machine. You can download it for your platform directly from the https://github.com/.
4. Authenticate yourself and your machine in addition to adding your machine's key to the github account by following steps 1 and 2 of the following links for :
   MacOS, Windows, or the set up Github directions for Ubuntu (linux).

Here are cliff notes for the last exercise. Note that these are general and the links above provide more OS specific directions.

1. Login to `https://github.com/` ,
2. on the top right corner, click on your avatar and select "settings",
3. on the left column, click on "SSH and GPG keys", and
4. click on "New SSH key" and follow the instructions Github provides from there.

[1]The Secure Shell Protocol (`ssh`) is a long-standing standard cryptographic network protocol for securely accessing devices over a newtork. It can be used to log into remote computers, and is the basis of commonly used file transfer protocols, like the secure copy protocol `scp`. The gist of how it works is that you generate what's called a "public-private key pair" on your computer using a program called `ssh-keygen` and RSA encryption. This creates two files: `id_rsa.pub` and `id_rsa`. The former is a public key which you have to upload to a remote server. The latter is your private key, which should *never* be shared outside of your computer. Both keys are just cryptographic hash strings. If you have a user account on a server, and it has your public key, you will be able to access it via `ssh` in a secure way from your computer which generated the key pair.

# 3 Lab day: Building your own repository

During this lab, you will get your repository set up and your toolbox from last lab uploaded to the online repository. There is more than one way to set up a repo. The technique in this lab is among the easiest.

## 3.1 Initialize a repo

We will set an example repo for our files, following the instructions below.

1. Login to `https://github.com/` ,
2. click on the button "New", on the top left corner, by the word "Repositories",
3. give the repo the name "testrep" and click on "Create repository".

## 3.2 Commonly used git commands

These are the most common git commands. They help keep the local and online repositories matching and allow you to easily document the changes made each time you upload files to the repositories. More details on some of these commands is provided in later sections.

- `clone`: Download and copy a repository to your machine that is new to your machine
- `pull`: Download a current version of your online repository online that the version on your machine is up-to-date.
- `add`: add new files or add changes to existing files in the repository to register them as files you want to track.
- `commit`: Take a "snapshot" of the changes to be made and add a comment for documentation. This is used to commit changes added with the `add` command.
- `push`: Send the local changes that have been added and committed to the online repository.

## 3.3 Download the repo for the first time

In order put files into our repo, we have to download a copy of it to our computer. The `git clone` command allows us to do this. First you must get the `ssh` address of the repo. Here are the directions for getting the address:

1. Go the website of the repository.
2. In the right side, there is green button called **code**. Click on it and then click on the **ssh** tab.
3. In the box, there is an address of the form `git@github.com:username/testrep.git`, where `username` is your Github login name.
4. Copy the text in that box.

**Make sure to choose the "SSH" address, as opposed to the "HTTP" version.**

Now you can go to your terminal and type the following command

```
git clone git@github.com:username/testrep.git
```

and hit enter. Your terminal should read as follows:

```
$ git clone git@github.com:username/testrep.git
Cloning into 'testrep'...
warning: You appear to have cloned an empty repository.
```

You have indeed cloned an empty repository. Now you will put some code into it. First, copy all the files you created in Lab 1 into the cloned file. The cloned repository directory will have the structure below.

```
testrep
└── Homeworks
└── Labs
    └── Lab 1
        └── lab1.py
```

Now you will put these files in the online repo. This involves three steps: `add`, `commit` and `push`. With the commit command, you can put a comment about the edits being uploaded in quotes at the end of the command. Also, with the commit command, we used the -a option which takes into account all modifications we have done. This updates/adds all changes to the online repository.

Change directory to `testrep`.

The commands for uploading the files to the online repo are:

```
git add * (or git add file1 file2 ...)
```

and

```
git commit -a -m "Added directories from lab 1"
```

and

```
git push
```

The results of these commands is as follows:

```
$ git add *
$ git commit -a -m "Added directories from lab 1"
[main (root-commit) 7d121c4] Added directories from lab 1
 1 file changed, 1 insertion(+)
 create mode 100644 APPM4600/Labs/lab1/lab1.py
$ git push
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Writing objects: 100% (6/6), 355 bytes | 355.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:username/testrep.git
 * [new branch]      master -> master
$
```

Now your local repository and the `github` repository are in sync, and all our changes have been saved. **Note: the homework file will move to the git repository when it is populated. i.e. You will see it in the git repository when you do your first homework assignment.**

Go to your github page where your online repository is at `https://github.com/username/testrep` (replace username with your Github login name. Your files that your just uploaded should be there. On top of the file list, the comment "Added directories from lab 1" you sent with your last commit should show along with the time at which the changes were saved.

## 3.4 Local vs. Remote changes

This document has been using the term "local" and "online"/"remote" to talk about the two linked repositories that we are maintaining. In this section, we will better detail what the difference is and use more formal language.

First, using the `git commit` command only saves changes locally on your computer. The changes are not uploaded until you execute the `git push` command. This means that the remote (online) server does not know about the changes on your machine at all. If you were to erase the local directory, all the changes you made after you pulled the repository would be lost regardless of how many `commits` you performed.

That is why it is very important to `push` every time we can to make sure that the remote repository is as much *in sync* with our local copy as possible, to avoid any issue coming from our computer that could result in data loss.

## 3.5 Repo copies

We might be interested in creating many copies of our repository and work on different things, and even on different computers. Once a push has been done from any of those copies, we can get those changes in any other copy, just by using the command below.

```
$ git pull
```

Of course, Git will refuse to overwrite changes, so we need to make sure to discard the changes we have in the repository we want to pull the last developments over. This can be done just by using the command below.

```
$ git stash
```

## 3.6 Reverting changes

Frequently we might have made changes on a file that we want to discard. This is done simply by using the command below.

```
$ git checkout file
```

Where `file` is the file we want to revert to the last version we saved in the remote repository. **The changes discarded are deleted forever**.

## 3.7 Commit selected changes

We have been using the command `git commit -a -m "comment"`, which commits **all** changes made in the repository. Just by changing the command to the one below, we can commit selected changes.

```
$ git commit -m "Comment" file1 file2 file3 ...
```

The same mechanism goes for the `add` command.

```
$ git add file1 file2 file3 ...
```

### 3.8 Best practices

- **Store text files almost exclusively.** Git is not able to optimize storing non-text files (i.e. photos, executables), they must be avoided to the greatest extent. Occassionally, but rarely, PDF files that do not change too frequently can be added.
- **Avoid storing unnecessary files.** If you store code that generates other files, make sure you only add the code files and not the generated files, this will keep your repository lean and clean.
- **commit and push VERY often.** Git stores changes to files, not all file versions, so it doesn't occupy more space to commit more times. It's better to be safe than sorry, you can always come back to previous versions if you saved them.

## 4 Exercises

*You will do these exercises on your own to improve on the skill set you just developed.*

1. If it is not already in your repo, at the root level, add an empty file named "README.md". **Then commit and push.**
2. Edit "README.md" by adding a quick explanation (minimum of 500 characters) of what this repo contains, what its subdirectory structure is like, and any other information you think would be useful for you in the future or for any prospective user of your repo. **Commit and push frequently, at least 5 times. Use comments explaining each commit.**
3. Check online if the changes are indeed reflected in the online version.
4. Go out of the `testrep` directory and **clone** the repo again with another name, using the command

   ```
   git clone git@github.com:username/testrep.git testrep2
   ```

   this will create another copy under the name `testrep2`. Check that your changes to "README.md" are there.
5. In `testrep2`, add at least 5 comments to a file of your choosing from Lab 1, explaining what the code does. After each comment, **commit and push**.
6. Go back to `testrep`.
7. **pull** from the online repository. Can you see the changes you did in `testrep2`?
8. Add a line at the top of "README.md" containing only your last name. **add, commit and push**.
9. Go back to `testrep2`, add a line to the top of "README.md" containing only your first name. **pull** from the online repository. Git should issue an error. With the tools you have learned using this Lab, how can you solve it?

## 5 Deliverables

For this lab you need to share your git repository with the TA and instructor at the username **APPM4600instruct**.

## 6 Further reading

By far, the best source of information is found in the online book **Pro Git**, accessible here: https://git-scm.com/book/en/v2. However, since Git is so widely used in the online community, a simple online search should be enough for most things.

## 7 Concluding remarks and advice

1. **Store text files only.** Version control systems are optimized such that the minimum amount of data is actually stored. This depends on the files being **text files**, such that the system can analyze them. If we need to use other kinds of files, such as PDFs or images, we need to either store them separately, or make sure that they do not change frequently.
2. **Commit and pull as much as possible.** Since the system will optimize the storage, there is no need to wait for committing and pushing. It's better to have stored a buggy code and have to go back to a previous commit than to lose a development because a computer failure.
3. **Store git mistakes you might have done.** If you committed and pushed the wrong files or changes, it is better to commit and push a new version reversing those changes than erasing the last commit. This way your development history is stored and the dangerous, data-losing habit of erasing commits is avoided. As discussed before, storage should not be an issue.
4. **Create repo copies at will.** Never be afraid of creating a new copy of the repo and experiment, only make sure to push meaningful changes to the repository, either to the `master` branch or any other. Copies are just copies, and should not contain important information that has not been pushed. As long as we push frequently, any copy can be created or eliminated at any point.

## A Branches

For your independent directory, you do not need to worry about anyone working on the files while you are working on them. For your project, you may run into this problem with collaboration. One option for avoiding this problem is to utilize the *branch* feature in git. This appendix provides an overview of branching for repositories.

## B Single branch development cycle

We will now use our repo in the simplest possible way, this is, just saving our changes in a single *branch*. A branch is the name `git` gives to a development timeline, we will explain further the meaning of a branch when making parallel developments in the next section.

For this, we `add` a new file named `README.md` containing only the line below.

```
This is line 1.
```

First we add the file, then we commit and push this change, this is done with the commands below.

```
$ git add README.md
$ git commit -a -m "Add line 1 to README.md"
[master 5c28ec6] Add line 1 to README.md
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 323 bytes | 323.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:username/testrep.git
   ef817d1..5c28ec6  master -> master
$
```

We now follow the cycle of modifying, committing and pushing two more times until `README.md` contains three lines as shown below (we only need to `add` once).

```
This is line 1.
This is line 2.
This is line 3.
```

We are going to use a new command now, that shows the last changes we have made in a timeline manner. The command is `git log`. For this, we add the flags `---all` that shows all the *branches* (we'll see what they are used for later), `---graph` shows a graphic depiction of the development process, and finally `---pretty="%h %s"` which will simplify our log such that we can read it better. The final command and output are shown below.

```
$ git log --all --graph --pretty="%h %s"
* d83b031 Add line 3 to README.md
* 5cea4c5 Add line 2 to README.md
* 5c28ec6 Add line 1 to README.md
* ef817d1 Added directories from lab1
$
```

Different outputs can be obtained by using different options after the `---pretty` flag. They can be found in the manual of this command, accessible using `man git-log`.

You can test that all changes have been saved by using the command `git checkout` and the number on the left of the output to `git log` called *hash*. For instance, `git checkout 5c28ec6` in our case, will set your local copy of the repo to one where `README.md` only contains one line.

Before continuing with the next section, make sure you execute `git checkout master` in order to come back to our last version.

## C Branches

We will describe now what is probably the most important feature of any version control system, *branches*.

Branches are development histories that may or may not end up being part of our software, they are developments that are initiated in a *sandbox* of sorts, and are only added to our code after we are certain that the developments are correct.

At this point we might be thinking: why wouldn't we just do it with the development cycle we saw in the previous section? The answer is that we want to track specific changes separately from other changes that we ourselves or any other member of our team might be doing on our main code.

Unbeknownst to us, up until now we have been working on a branch, this branch is called `master` and all completely approved changes are committed to this branch. This can be done differently, but explaining other development cycles is not important at this time.

To create a branch we execute the command `git branch newdev`, where `newdev` could be any name we want to give to this branch. It is recommended to give short names to branches that are meaningful, to improve navigation inside the repo. After this command we `checkout` the branch by using the command `git checkout newdev` as shown below.

```
$ git branch newdev
$ git checkout newdev
Switched to branch 'newdev'
$
```

All the changes we now commit and push will be stored separately from our main code. To test this, we modify the first line of `README.md` to read as shown below.

```
This is line 1 modified in branch newdev.
This is line 2.
This is line 3.
```

After this, we commit and push, but modify slightly our push command to be as shown below.

```
git push --set-upstream origin newdev
```

We do this only once, to tell `git` that our branch has been originated and, in a way, will *follow* the `master` branch where it was originated. After this we continue just using `git push` when we want to store changes to branch in the remote repository. The input and output should look as shown below.

```
$ git commit -a -m "Modified line 1 of README.md"
[newdev 3387535] Modified line 1 of README.md
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push --set-upstream origin newdev
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 363 bytes | 363.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'newdev' on GitHub by visiting:
remote:      https://github.com/username/testrep/pull/new/newdev
remote:
To github.com:username/testrep.git
 * [new branch]      newdev -> newdev
Branch 'newdev' set up to track remote branch 'newdev' from 'origin'.
$
```

We now go back to the `master` branch by using the command shown below

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
$
```

We just went back to a version of our code where `README.md` looks as shown below.

```
This is line 1.
This is line 2.
This is line 3.
```

We modify `README.md` to read as shown below.

```
This is line 1 modified from master branch.
This is line 2.
This is line 3.
```

We commit and push. By doing this, our `master` branch has drifted from the `newdev` branch, as shown using `git log`.

```
$ git log --all --graph --pretty="%h %s"
* b285e8f Modified line 1 of README.md
| * 3387535 Modified line 1 of README.md
|/
* d83b031 Add line 3 to README.md
* 5cea4c5 Add line 2 to README.md
* 5c28ec6 Add line 1 to README.md
* ef817d1 Added directories from lab1
$
```

## D Merging a branch

We now `checkout` the `newdev` branch by using the commands below.

```
$ git checkout newdev
Switched to branch 'newdev'
Your branch is up to date with 'origin/newdev'.
$
```

Where the file `README.md` should read as shown below.

```
This is line 1 modified in branch newdev.
This is line 2.
This is line 3.
```

We modify the file to read as shown below.

```
This is line 1 modified in branch newdev.
This is modified (in newdev) line 2.
This is line 3.
This is line 4.
```

We commit and push using the commands below.

```
$ git commit -a -m "Modifications in order to merge"
[master eb2e026] Modifications in order to merge
username@computername:~/workspace/testrep$ git push
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 417 bytes | 417.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To github.com:username/testrep.git
   b285e8f..eb2e026  master -> master
$
```

We now want to bring all the changes made in the branch `newdev` to the `master` branch. For this, we `checkout` the master branch, and we execute the command `git merge newdev`, we should obtain the output below.

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
$ git merge newdev
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
$
```

This means that `git` tried to merge the changes between those branches and failed to uniquely identify what changes should be kept. Usually the changes are just lines and it is able to do it automatically, otherwise we need to do it manually.

Our `README.md` file now reads as shown below.

```
<<<<<<< HEAD
This is line 1 modified from the master branch.
This is line 2.
This is line 3.
=======
This is line 1 modified in branch newdev.
This is modified (in newdev) line 2.
This is line 3.
This is line 4.
>>>>>>> newdev
```

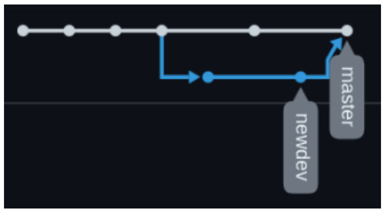
We immediately identify the changes we made in different branches. Now we have to decide what changes to keep or discard. We modify `README.md` to look as shown below.

```
This is line 1 modified from the master branch.
This is modified (in newdev) line 2.
This is line 3.
This is line 4.
```

We commit and push, after which `git log` shows the output below.

```
*   eb2e026 Modifications in order to merge
|\
| * cc37f61 Modified lines 2 and 4
| * 3387535 Modified line 1 of README.md
* | b285e8f Modified line 1 of README.md
|/
* d83b031 Add line 3 to README.md
* 5cea4c5 Add line 2 to README.md
* 5c28ec6 Add line 1 to README.md
* ef817d1 Added directories from lab1
```

Which shows the changes we have made in a timeline manner, such that we can clearly identify the versions we created and come back to any of them to see how the changes were performed. If we go to `github` at `https://github.com/username/testrep`, then click on `Insights` and then on `Network`, we should see the image below.



This process of branching and merging, allows us to perform changes and track them, such that the history of our development is stored and we can come back to any point in our development. This is useful when finding bugs, older developments or results.

## E Concluding remarks about branches

- **Branches do not necessarily have to be merged.** Create branches at will, even for the smallest of developments that merit to be separated from the main code. Those branches will only occupy a small space and can be deleted in the future.

- **Branches can have branches and be merged.** The command `git push --set-upstream origin newdev` sets the branch `newdev` to follow the branch where it was originated, regardless of it being the `master` branch. Branches can have sub-branches, sub-sub-branches and so forth. Everything can be discarded or merged to the `master` branch at a future point in time. The following commands create a branch from another branch and store that information in the remote repo.

  ```
  $ git branch newdev2
  $ git checkout newdev2
  ... develop something and commit changes to the branch newdev2...
  $ git push --set-upstream origin newdev
  $ git branch newdev3
  $ git checkout newdev3
  ... develop something and commit changes to the branch newdev3...
  ```

```
$ git push --set-upstream origin newdev3
... output ...
```

We can now checkout `newdev2` and merge `newdev3` into the branch `newdev2` just by using `git merge newdev3`, or checkout `master` and merge into it using the same command `git merge newdev3`. Following a `master` branch is a typical development choice but there is nothing special about this particular branch.