The Science of Computing III **Living *with* Cyber**

Lesson: Git for beginners **Beam**

Most of the programs you've written in the curriculum so far have probably been written on your own computer by yourself with just a single copy. If you had to keep track of multiple copies/versions, you probably used some ingenious naming convention e.g. `mycode.py, mycode_draft2.py, mycode_finaldraft.py, mycode_submitted.py`, etc. However, as you tackle larger and larger projects, there is often a need to use a *distributed version control system*. A distributed version control system is just a way of keeping track of different versions of a project spread across multiple systems/computers and the most common such system is called **git**.

On a single system, git can be used to track changes in any files of interest i.e. inform the user as to what differences have been made since it was last saved, allow you the possibility to revert to older versions of the file, etc. This is particularly helpful if you add a feature to your code/files that breaks everything and you want to revert to an older version of your code/files. Git will allow you to go back to any version of your files since the time you gave it permission to track them regardless of how far back that might be.

When used with a central server to which other programmers have access, git can then be used to coordinate work among your colleagues i.e. allow each person work on copies of the same code without fear of messing it up for everyone else. Final copies/versions of code can then be submitted to the central server so that one's colleagues have access to those files (and their history). Git will also (as best it can) automatically merge changes to files made by different colleagues which allows them to work on different parts of the same code at the same time.

When used with a central server, Git can also be used to keep track of an individual's projects if they are working across multiple systems e.g. a work computer and a personal computer. This allows them to easily access the latest version of a file regardless of which computer they are using.

There are a number of servers that are designed to facilitate the collaboration component of git as well as allow remote storage of your projects: the most common of which is **github**. Others include **gitlab**, **bitbucket**, etc. Most are free to use as long as you get an account online and are open to the possibility of your code being public i.e. anyone with the link can view and download your code. If you are concerned about security, there are options to get private accounts some of which come at a price.

To get the most out of git, one will have to get comfortable with using the command-line but there are a lot of git GUI clients that can be used as well. It is suggested that you look at a few before making a decision on which client to use. Some work better with different operating systems, and some cost a little money so take your time to look up a few and find the best for you. For example: **Github desktop** is a client git GUI but is only available for Windows and Mac and only if you are using Github as your server. Another alternative is **GitKraken** which can work on windows, Mac and Linux and can be integrated with either Github or Bitbucket as your server.

For this lesson, we shall be using the command line and Github as the server. If you are using a Windows system, **git bash** can be used to allow you to have a terminal-like experience when working with git. Even if you decide to download and install a git client, following this lesson in the terminal will help you grasp what the different commands are, what they mean, and how to use them.

Installing Git.
To install git on your linux computer (e.g. your raspberry pi), run the following command in your terminal

```
sudo apt install git-all
```

And that's it.

For windows systems, you might want to look into the installation process for git bash or some other git client alternative. As an example, https://gitforwindows.org/ provides a git client GUI and terminal that can be installed and used but there are other git options. Either way, Google is your best friend.

Git for individuals.
As a single user, git can be used for version contol i.e. to keep track of different changes to your code/files and allow you to revert to whatever older versions of your code/files. It will also allow you download code from publicly available sources (on a server like github) and then make your own local adjustments to that code.

If you have the address of a project on github (or any other git server) that you want to make a local copy of, the command for that is

```
git clone <ADDRESS>
```

for example:

```
git clone https://github.com/ankunda/WelcomeToGit.git
```

This will copy the project on github to your computer making a folder in the process to help with organizing it in a similar manner. Even though we are highlighting this in the "individual" section of git, this command is important to note because it is the same command used initially to create a copy of some remote folder that you will be working on.

Alternatively, if you'd like to start a project from scratch with your own local code, you could create and use a local folder/directory in which the files that you want to keep track of are stored.

```
mkdir mygitfolder
cd mygitfolder
git init
```

The commands above create a folder called *mygitfolder*, navigate into that folder, and then initialize git to start keeping track of the files in that folder.
Once git is properly initialized (whether using the clone or init commands), there should be a secret folder within your directory (mygitfolder) called **.git**

```
anky@jbex:~/Desktop/code/mygitfolder$ git init
Initialized empty Git repository in /home/anky/Desktop/code/mygitfolder/.git/
anky@jbex:~/Desktop/code/mygitfolder$ ls
anky@jbex:~/Desktop/code/mygitfolder$ ls -al
total 12
drwxr-xr-x  3 anky anky 4096 Nov  7 16:33 .
drwxrwxr-x 33 anky anky 4096 Nov  7 16:33 ..
drwxr-xr-x  7 anky anky 4096 Nov  7 16:33 .git
anky@jbex:~/Desktop/code/mygitfolder$
```

Even with the folder set up as we have shown thus far, git will still need to be given explicit permission to keep track of and manage specific files.
As a demonstration, create a python file in mygitfolder that we'll keep track of using git. It could be called HelloWorld.py. All it needs to have for now is a print statement that prints out "hello world."

```
anky@jbex:~/Desktop/code/mygitfolder$ ls -al
total 16
drwxr-xr-x  3 anky anky 4096 Nov  7 16:35 .
drwxrwxr-x 33 anky anky 4096 Nov  7 16:33 ..
drwxr-xr-x  7 anky anky 4096 Nov 14 09:14 .git
-rw-r--r--  1 anky anky   21 Nov  7 16:35 HelloWorld.py
anky@jbex:~/Desktop/code/mygitfolder$
```

A command to use at ANY time during this process is

```
git status
```

The **status** command can be used to inspect the state of the folder that git is keeping track of. It typically also provides suggestions for other git commands that can be executed in your workflow.

```
anky@jbex:~/Desktop/code/mygitfolder$ git status
On branch master

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
anky@jbex:~/Desktop/code/mygitfolder$
```

The output above shows that we are using the master branch (i.e. the original copy of the folder/project), that there aren't any changes that we have committed to (referred to as commits). It also recognizes that there is a new file in the folder – a file that it hasn't yet been given permission to keep track of.
If we want to add this file to the list of things for git to keep track of, we use the **add** command.

```
git add <filename>  # to add a file specifically OR
git add .           # to add the entire current directory and all its
                    # contents. Note the period after the add command
```

```
anky@jbex:~/Desktop/code/mygitfolder$ git add HelloWorld.py
anky@jbex:~/Desktop/code/mygitfolder$ git status
On branch master

No commits yet

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

anky@jbex:~/Desktop/code/mygitfolder$
```

We can now see that the status has changed to show that there is a new file that it is keeping track of. You can make as many changes to the file as you would like. The changes could even potentially span multiple days, and the changes might even involve adding extra files to the folder. You will need to repeatedly run the git add command to allow git to keep track of those changes. Right now this file has been added to the "stage area" which is an intermediate area in git that files are added to before being committed (or saved).

Moving the files and/or changes out of the stage area will involve git commit. Committing allows to clearly mark a point in the project timeline that git can potentially revert to in the future. Each commit will require a descriptive message to be associated with it so that it is easy for you or any other collaborators on the project to identify what each commit point was related to in your project timeline.

```
git commit -m "short descriptive message"    # OR
git commit          # this will automatically open an editor that
                    # will allow you to write a much longer
                    # description about this point in the project
                    # timeline.
```

Because committing is such a central part of the git process, git will ask for some credentials during your first project commit. It will then use those credentials as your identifier during all future commits in the project timeline. The commands to configure your git identifier are shown below. Make sure to replace EMAIL and FULL NAME with your actual email and full name.

```
git config --global user.email <EMAIL>
git config --global user.name "<FULL NAME>"
```

```
anky@jbex:~/Desktop/code/mygitfolder$ git config --global user.email kiremire@latech.edu
anky@jbex:~/Desktop/code/mygitfolder$ git config --global user.name "Ankunda Kiremire"
anky@jbex:~/Desktop/code/mygitfolder$
```

And now you're ready to commit your changes using the **commit** command.

```
git commit -m "a short message describing your adjustments"
```

```
anky@jbex:~/Desktop/code/mygitfolder$ git commit -m "First draft of hello world"
[master (root-commit) 6c46f6d] First draft of hello world
 1 file changed, 1 insertion(+)
 create mode 100644 HelloWorld.py
anky@jbex:~/Desktop/code/mygitfolder$
```

When using git as an individual, `add`, `commit` and `status` will be the commands you use the most. Occasionally, however, you will need to go back to an older version of the code. This might be because you made some changes that broke your code or project and are not sure what changes exactly those were. When such a need arises, remember that you can only revert to the state your file was at at any commit point in your project timeline.

To see all the possible revert points, use the `log` command. The git log command will list all commit points of the project starting with the most recent commit. For each commit point, it will show a hash[1] of the commit point, as well as the user who committed, when it was committed, and what descriptive message was associated with that commit.

```
git log
```

The result screen of the log command should show you the need for good descriptive messages during a commit, as well as the need for frequent committing particularly when you get to a good stopping point during your work session in your project. The more commits you have, the more places you can revert to

[1] A hash is a unique string used to differentiate between different files without opening the files. This git hash algorithm gives a 40 character string that is made up of hexadecimal characters.

should something go awry in your project. There is of course a price to be paid if you commit too much and that is you spend a lot of time creating and updating your commits instead of doing work on the actual project. The size of the git folder might also take up more space with too many commits.

To revert to an older version of your project, use the provided hash string as the argument to the command below

```
git checkout dc3c787c337ae61d71d25684ec022ddd820c5733
```

You can go back to the original version of the file (i.e. the current point in the timeline) by using the checkout command with master as the argument.

```
git checkout master      # to go back to the present point in your
                         # project timeline
```

Git commands so far.

```
git init  # to intialize the folder. Only needs to be done at the
          # beginning of the project.
git clone <address> # to make a local copy of a remote repository.
git add <filename>  # to allow git to track the file
git add .      # to allow git to track all files in folder.
git commit -m "message"  # to mark a point in your project timeline
git log   # to see all points in your project timeline
git checkout <hashvalue> # to go back to a point in your timeline.
```

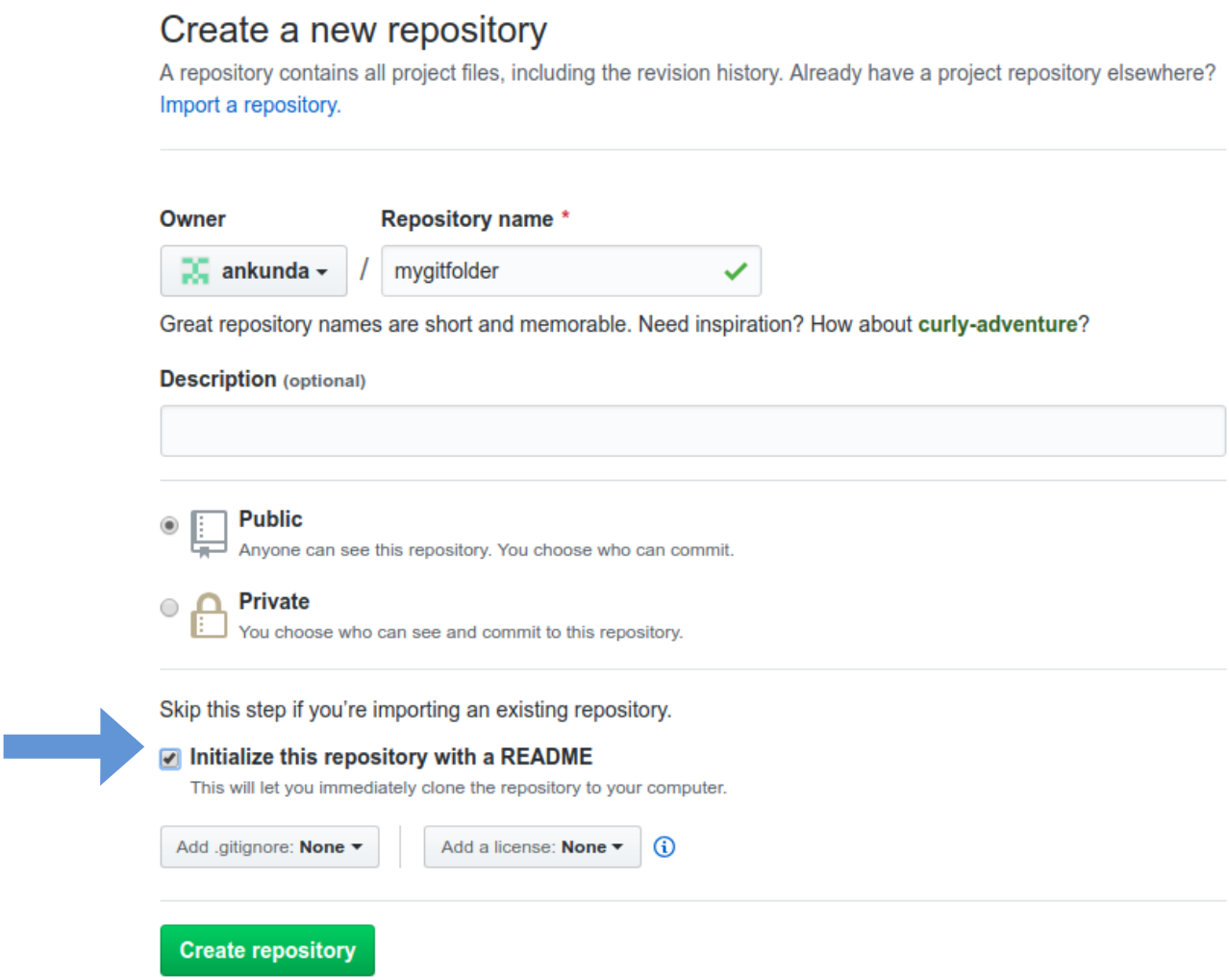
Git for teams

Git interacts well with online repositories which allows for teams to work on the same project without the tediousness of sending each other and keeping track of the latest versions of projects and the files therein. As mentioned earlier, git and github can also be used by a single user to keep track of the same project across their multiple computers/systems.

We shall now walk through a sample process for creating a remote repository with a local copy i.e. an online copy of a folder on your computer. The easiest way to do this is to

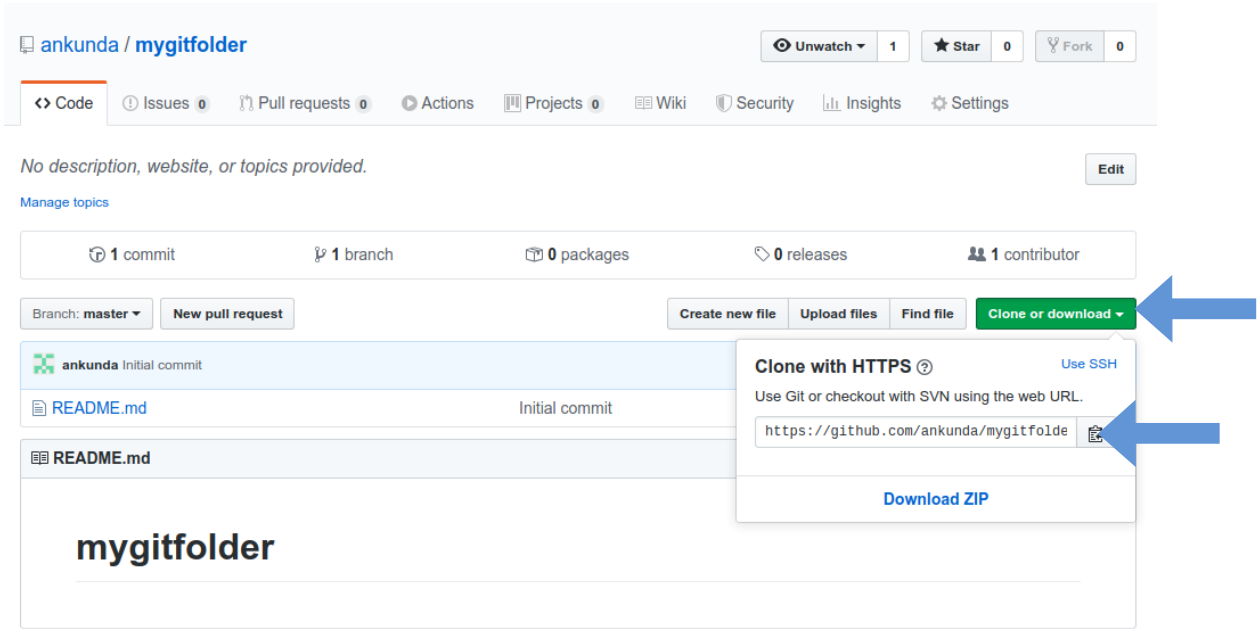
- create an empty remote repository,
- clone that empty repository on your own computer,
- update your local copy with the files you want to keep track of, and then
- send those updates to the remote repository.

To create a repository on github, create a github account, and then select "new repository" at the top left portion of your github page. Make sure to select the option that initializes the repository with a readme file. There are a few other options that you can choose such as what the repo starts with and who can see it.

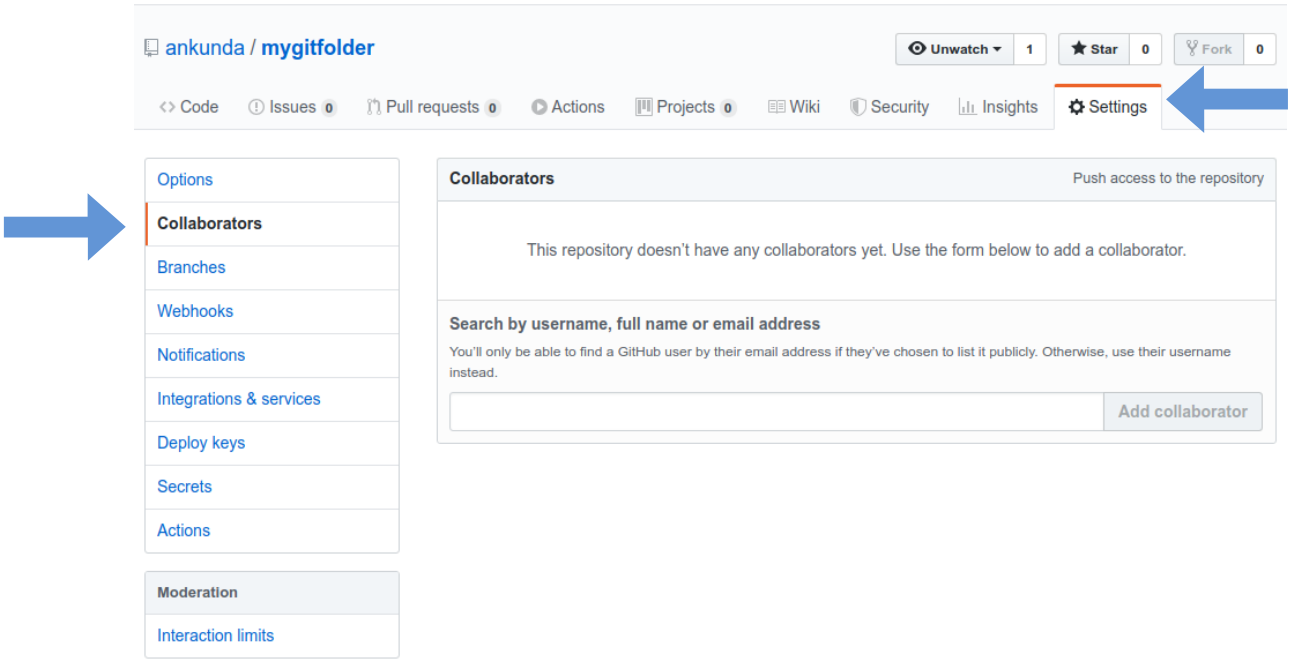


Once a repository has been created, you should have access to a url that you (or anyone else) can use to update their own local versions of the project. Recall that the command to clone a remote repo is

```
git clone <ADDRESS>
```



If the creator of the repo would like for their team members to contribute to the project, they should add collaborators to the project using their teammates github usernames/credentials. This can be done under the collaborators option of the settings tab.



Once cloned, the local copy of the folder should only have the readme file and nothing else. The team lead (or any member of the team) can now add files to the folder and commit those changes.

```
git clone <address>      # to create a local copy of the repo.
# One member of the team should add any files to the folder and then
# commit those changes.
git add .
git commit -m "initial files"
```

Once you are ready to send the updates to the remote repo, use the push command.

```
git push       # use it with the default settings OR
git push <source-name> <name-of-branch> # specify the source-name
               # which is typically "origin" and name-of-branch
               # which is typically "master"
```

The first time that you push, you will be asked for your github credentials. Historically, one could use just a username and password. However, to improve security, github now uses personal access tokens in place of passwords. The owner of the github repo gets to create as many personal access tokens as they'd like. They work the same as a password except that they have a time limit, and the owner can limit the tasks that can be accomplished with the token. Additionally, the owner of the repo can revoke any personal access token at any point.

To get a token you will need to take the following steps.

1. **Verify your email.** Log into your github account. This will in all likelihood require typing in a verification code sent to your email.

2. Select **Settings → Developer Settings → Personal access tokens.** Settings can be accessed in the drop down menu next to your account image in the top right part of the home page.

3. **Generate new token.** You will need to provide some note for its description, how long the token will be valid for, and what the token can be used for. For this demonstration, just selecting **repo, admin:repo_hook,** and **delete_repo** should be adequate.

4. **Copy the provided token and use it** in place of the password when asked for your github password while in terminal. You might need to use **Ctrl+Shift+V** to paste in terminal.

Once you have a username and token, execute the push command.

```
anky@jbex:~/Desktop/code/mygitfolder$ git push origin master
Username for 'https://github.com': ankunda
Password for 'https://ankunda@github.com':
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), 314 bytes | 314.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/ankunda/mygitfolder.git
   3475c1e..bc95efe  master -> master
```

Once the changes are pushed to the remote repo, any colleague can download the latest copy of the folder/file by executing the pull command.

```
git pull       # use default source and branch OR
git pull <source-name> <name-of-branch> # specify.
```

Once executed, all group members should have local copies of the folder created and pushed by the first team member. Congratulations – You have created a repo, and all your group-mates have access to that repo.

```
Remember the commands we are discussing are only done to start or to
update the remote repo. You can edit files, add files, etc using your
usual methods.
git pull  # to get latest copy of the repo (done before you start any
          # work or adjustments)
# make any adjustments to your files/folders during this phase.
git add . # to add any adjustments to your local git history
git commit -m "message"  # to mark a point in your project timeline
git push  # to push the changes to the remote location
git status # to get helpful messages and descriptions.
```

Below are some screenshots of what your screen might look like after executing the commands above.

git status command after executing an **add** and **commit**.

```
anky@jbex:~/Desktop/code/mygitfolder$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

git push command (as suggested by output of **status** command above)

```
anky@jbex:~/Desktop/code/mygitfolder$ git push origin master
Username for 'https://github.com': ankunda
Password for 'https://ankunda@github.com':
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), 314 bytes | 314.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/ankunda/mygitfolder.git
   3475c1e..bc95efe  master -> master
```

git pull command if nothing was changed on the remote repo.

```
anky@jbex:~/Desktop/code/mygitfolder$ git pull origin master
From https://github.com/ankunda/mygitfolder
 * branch            master     -> FETCH_HEAD
Already up to date.
```

git pull command if file was changed on the remote repo in the time since your last **pull** command

```
anky@jbex:~/Desktop/code/mygitfolder$ git pull origin master
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/ankunda/mygitfolder
 * branch            master     -> FETCH_HEAD
   bc95efe..5c70194  master     -> origin/master
Updating bc95efe..5c70194
Fast-forward
 HelloWorld.py | 1 +
 1 file changed, 1 insertion(+)
```

Notice that it informs you that only one file was changed, and what those changes look like. In this case only one line was inserted into the original file.

Demonstration of a typical process.

To demonstrate the typical process of using git and github, we are going to work on a small programming project in groups. Prior to kickoff, every student should create a github account if they haven't already done so. As a suggestion, select a username that you won't be embarrassed to share with a future employer when they ask to see your github repository.

1. Each group should select a leader. That leader will have the task of creating the initial repository on github that everyone will be working off of. Once the repository is created, he/she should add the group members as collaborators to the repository and distribute the github url to everyone in the group (e.g. via email or text message) so that everyone can participate during the course of the demonstration. Commands you might use include **clone** and **status**

```
git clone <address>
```

At this point, every member of the group should have a copy of the repository on their own systems. It will probably only have the readme file in it.

2. After a short discussion with the group members, the leader should create an initial python file that has the skeleton of a project that they would all be interested in adjusting. It doesn't have to be particularly involved. It could be as simple as a hello world program, or a program with a few stubbed out functions. An example is provided below.

```
def student1Function():
     pass

def student2Function():
     pass

     ...

def studentNFunction():
     pass

############ main program ##########
student1Function()
student2Function()
...
studentNFunction()
```

3. The leader will them commit this newly created file to the remote repository. Commands you might use include **status**, **add**, **commit**, and **push**. Once a source-name and name-of-branch have been used in a push command, subsequent push commands do not need to explicitly mention them again unless you wish to change either the source-name or branch-name.

```
git status
git add <filename> # or git add .
git commit -m "<descriptive message>"
git push <source-name> <branch-name> # or just git push
```

4. **Taking turns**, all the other group members should now proceed to update their local folders so that they can access the newly added file. Commands they might use include **status**, and **pull**. Similar to the push command above, git pull will be adequate once you've used the full git pull command before with the source-name and branch-name. Once a member has pulled the repo, they can make any adjustments to the file, and then push those changes to the repo. Make sure only one member is doing this step at a time i.e. a team member pushes their changes before another team member pulls the file.

```
git status
git pull <source-name> <branch-name>
# make any adjustments to the file/folder
git add .
git commit -m "<descriptive message>"
git push
```

While the process you just went through is the typical process and is one you'll experience the vast majority of the time, there is one exception that you will need to know how to navigate.

When conflicts happen

There are times when two or more members of the same team make changes to repo at the same time. This might happen when each member makes changes to:

1. a different file in the same repo,
2. different parts of the same file on the same repo, or

3. the same part of the same file on the same repo.

In any of those cases, git will only recognize a problem when the second person attempts to push their changes. This is because the changes they will be trying to push are based off of a now outdated version of the original file. When this happens, we are dealing with what is referred to as a conflict.

Git will try to deal with a conflict automatically. Initially, it will warn you that you have an outdated file and recommend that you execute a **pull** again. It will then try to automatically combine the updated remote repo copy with your local copy. The automatic combination is pretty effective if the changes were either made to different files, or if the changes were in different parts of the same text file[2].
In those cases, all the user will have to do is push again so that the newer combined version is sent to the remote repo.

However, if the changes were made to the same region (e.g. same lines) of the same file, then it will have to use a human to make the combination and you (the second team member) are that person. Git will change the file and put markers in it showing the portion of the file and the two versions of it that it was unable to automatically combine. It is the responsibility of the human (the second team member) to recognize the conflict regions, edit them till it makes sense, and then **add**, **commit**, and **push** those changes to the remote repo.
Note that **git status** can be used at any point in this process for pointers on what commands could be executed next.

[2] Most programming source code is written in a text file and can therefore be automatically merged by git. However, files with formatting e.g. microsoft word documents, pdfs, etc are not text files and therefore cannot be automatically merged. This means that if two people make changes to a microsoft word document, git will not be able to merge it. It will be the responsibility of the second user to download a fresh copy of the original document, and then copy and paste his own changes into it, then save it, and push it.

The diagram below shows an example of a file with some conflicts. One user put line 10 inside the function stud3(), while another put line 12. They can't be automatically reconciled so that function should be adjusted before the next commit. Any extra lines filled with conflict information should also be deleted since they are not part of the project.

```
 1 def stud1():
 2     print("Hello World")
 3     print("_" * 50)
 4
 5 def stud2():
 6     print("something else")
 7
 8 def stud3():
 9 <<<<<<< HEAD
10     print("_" * 23)
11 =======
12     print("*" * 34)
13 >>>>>>> db873812d1b5ca76db8fb4d734dd8bfde6e97c2e
14
15 def stud4():
16     pass
17
18 def stud5():
19     pass
20
21 stud1()
22 stud2()
23 stud3()
24 stud4()
25 stud5()
```

Once that is done, you'll want to **add**, **commit**, and then **push**.

```
git add -A
git commit -m "merged some conflicts with stud3()"
git push
```

I suspect that you'll know this by the time you're done with the demonstration above but as a rule of thumb,

1. **its always a good idea to pull the latest copy of a branch before you begin ANY work on it**,
2. **commit** any changes you make regularly, and
3. **push** all changes before you finish your work on the project for that session.

Clear division of labor will help with minimizing conflicts especially the ones that can't be automatically merged by git. But even then, there are going to be times where a conflict is unavoidable and you should be prepared to deal with it.

```
git pull # to get the latest version of the repository
git add -A # to add all files in the folder to the stage area
git commit -m "<message>" # to save those changes and get them ready
                # to be sent to the remote server
git push  # send my version of the folder to the remote server
git status     # what's the condition of all my files
```

Ignoring some files.
Occasionally there might be some files that you don't want to be tracked and sent to the server by git. This could be for any number of reasons e.g. they are system specific and therefore wouldn't benefit anyone else in the group, or maybe they will and should be created on the fly by anyone contributing to the project and therefore it doesn't make sense to send your copies. Whatever the reason, the way to inform git that some files should not be tracked and committed is by using a "**.gitignore**" file. Note the period at the beginning of the file name which in the linux world means the file is hidden/secret. To create such a file, execute the following command while in the folder that git is keeping track of.

```
touch .gitignore
```

This should create an empty file with the appropriate file name. You can now open that file using your favourite editor e.g. vim, nano, leafpad, etc. and put the names of the files that you want git to ignore in it making sure to put every filename on a new line. Some examples are shown below.

```
*.class
*.pyc
myownnotes.txt
```

The *.class refers to every file in the folder that ends with ".class" i.e. all class files (which might be a result of compiling a java file). Similarly, the *.pyc refers to every file in the folder that ends with ".pyc" i.e. all python compiled files. These are just suggestions so feel free to populate your .gitignore file with whatever you'd like. If you don't have any files to ignore, then either leave the .gitignore file empty or don't create one to begin with. Note that you should create and populate the .gitignore file before you add any of the files you'd like to ignore to the stage or push them to the server the first time. That way, git will know to ignore them from the beginning.

Making another branch.
A branch is a path of development for your project and its files. Right now we are using the main branch (called master) and all changes are committed to that branch. However, there will occasionally be a need to work on a feature of your project that shouldn't be merged with the original version while your feature version is still undergoing its own changes. In such a situation, the person making the feature will have to make another branch as opposed to working off of the main branch. Creating a branch makes a copy of master and allows you (or anyone else) to make as many changes as you'd like to that branch until it is completed before that branch (and the new working feature) is merged to the original master branch (that other people might have been working on in the mean time).

To create the branch the first time use and switch to it.

```
git checkout -b <branch-name>
```

fyi, the branch-name cannot have spaces in it. The **git status** or **git branch** commands can be used to let you know what branch you are currently working on.

```
anky@jbex:~/Desktop/code/myothergitfolder/mygitfolder$ git checkout -b newFeature
Switched to a new branch 'newFeature'
anky@jbex:~/Desktop/code/myothergitfolder/mygitfolder$ git status
On branch newFeature
nothing to commit, working tree clean
anky@jbex:~/Desktop/code/myothergitfolder/mygitfolder$ git branch
  master
* newFeature
anky@jbex:~/Desktop/code/myothergitfolder/mygitfolder$
```

The new branch will have to be pushed upstream to the remote server so that other users can access it the first time its created. Once its on the server, any person can get to it by using the git checkout command.

```
git checkout <existing-branch-name>
```

When you are done with all the adjustments to the new feature branch and are ready to combine it with master, switch to the master branch, and then execute the merge command.

```
git checkout master
git merge <existing-branch-name>
```

Once the merge is successful, you can then delete the branch.

```
git branch -d <existing-branch-name>
```

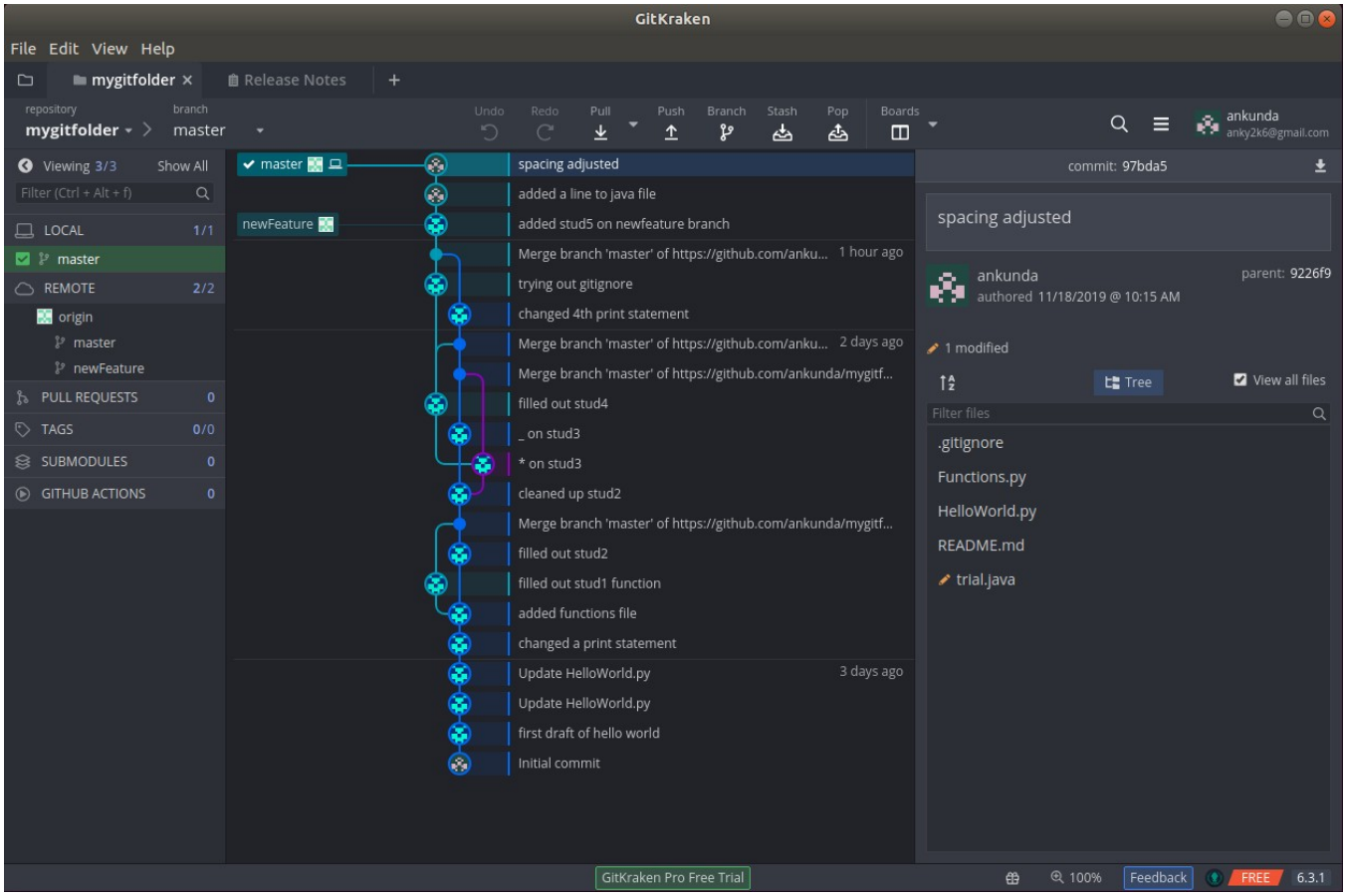
```
anky@jbex:~/Desktop/code/myothergitfolder/mygitfolder$ git branch
  master
* newFeature
anky@jbex:~/Desktop/code/myothergitfolder/mygitfolder$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
anky@jbex:~/Desktop/code/myothergitfolder/mygitfolder$ git merge newFeature
Updating e6e3f8d..b53adff
Fast-forward
 Functions.py | 5 ++++-
 1 file changed, 4 insertions(+), 1 deletion(-)
anky@jbex:~/Desktop/code/myothergitfolder/mygitfolder$ git branch
* master
  newFeature
anky@jbex:~/Desktop/code/myothergitfolder/mygitfolder$ git branch -d newFeature
Deleted branch newFeature (was b53adff).
```

And that is a very basic very quick introduction to git. It might seem like a lot but it gets easier with practice. I don't expect you to get everything the first time (even I end up googling a lot of things every time I use git and github), but I expect it to get easier with time, and you'll find that it helps a lot with coordinating and synchronizing your code within your group and that will be particularly helpful as you tackle the group project you have this quarter.

Git GUI client.

As a final note, for those of you who are completely uninterested in practicing and mastering the command line interface for git, there are multiple GUI clients that can be used. A GUI means that instead of typing out commands, you'll have to click buttons and understand the layout of the program. I typically find this slower than knowing commands, but to each his own.

Take your time finding an appropriate GUI client if this is what you want to use. Some will work on specific operating systems, some aren't free, and some will have more intuitive layouts than others. As an example, I looked into gitkraken (which can be found at https://www.gitkraken.com/). Note that even with a GUI you'll still have to take some time to get comfortable with the way the buttons are arranged, and what the process is for working with that client, and perhaps some keyboard shortcuts to get faster. Unfortunately, walking you through the process of getting comfortable with the GUI client is beyond the scope of this lesson, but I am sure that if you understand the terms and commands that were used during the command-line demonstration earlier, you should be able to figure your way around any gui client.


GitKraken
File Edit View Help
mygitfolder
Release Notes
repository
mygitfolder
branch
master
Undo
Redo
Pull
Push
Branch
Stash
Pop
Boards
ankunda
anky2k6@gmail.com
Viewing 3/3
Show All
Filter (Ctrl + Alt + f)
LOCAL 1/1
master
REMOTE 2/2
origin
master
newFeature
PULL REQUESTS 0
TAGS 0/0
SUBMODULES 0
GITHUB ACTIONS 0
master
newFeature
spacing adjusted
added a line to java file
added stud5 on newfeature branch
Merge branch 'master' of https://github.com/anku... 1 hour ago
trying out gitignore
changed 4th print statement
Merge branch 'master' of https://github.com/anku... 2 days ago
Merge branch 'master' of https://github.com/ankunda/mygitf...
filled out stud4
_ on stud3
* on stud3
cleaned up stud2
Merge branch 'master' of https://github.com/ankunda/mygitf...
filled out stud2
filled out stud1 function
added functions file
changed a print statement
Update HelloWorld.py 3 days ago
Update HelloWorld.py
first draft of hello world
Initial commit
commit: 97bda5
spacing adjusted
ankunda
authored 11/18/2019 @ 10:15 AM
parent: 9226f9
1 modified
Tree
View all files
Filter files
.gitignore
Functions.py
HelloWorld.py
README.md
trial.java
GitKraken Pro Free Trial
100%
Feedback
FREE
6.3.1