Lab 0: Git

Due: Tuesday, July 3rd, 11:59pm

***NOTE**: This lab is based on a Linux and Git lab originally written by Anne Rogers for CMSC 12100 (and updated and edited by a variety of instructors and TAs throughout the years)*

In this week's lab, you will learn about Subversion (SVN) and Git to handle revision control (or *version control*) of your projects and assignments, and to submit your work for grading. In this class, we will mostly be using Git, the more widespread version control system that is more commonly used in a variety of software projects. Nevertheless, for those of you that are curious, I've also included a basic introduction to SVN which you should also complete once you set up your repository.

Like SVN, Git is a *version control system* that maintains files in a *repository* that contains not just files, but also a record of all the changes made to those files. Git tracks every version of a file or directory using *commits*. When you have made changes to one or more files, you can logically group those changes into a "commit" that gets added to your repository. You can think of commits as "checkpoints" in your work, representing the work you've done since the previous checkpoint. This mechanism makes it possible to look at and even revert to older versions of a file by going back to your code as it was when you "checkpointed" it with a commit.

You will each make a personal Git repository. You may have heard of GitHub, a web-based hosting service for Git repositories. We will be using GitHub for the course project and labs.

Some basic commands

1. Open the terminal or command prompt (You need to get Cygwin installed if you're using windows, makes linux native commands work just as well on your Operating System. Ask me if you need any help doing this, especially adding cygwin to your path variables.)
   Change to the root directory by typing cd /.
   Type pwd and ls and have a look around.
   Type cd with no arguments. Type pwd to see where you are.
   Try cd .. and pwd.
   Try cd - and pwd.
2. Type cd with no arguments.
   Type ls to see what's there.
   Type ls -1 (that's the numeral 1) to display the contents of the working directory, one per line.
   Notice the difference between the output of ls and the output of ls -F.
   You can combine the effects of the options 1 and F in several ways: try ls -1 -F, ls -F -1, ls -1F and ls -F1.
   Try ls -lFG.

There are many options to the ls command. Type man ls and read about some of them. Type q to exit man.

3.

Change to your home directory using cd. (From now on, any instructions that direct you to change directories mean you should use cd.)
Create a batch of empty files by typing touch file1.empty file2.empty file3.empty file4.empty file5.empty.
Type ls *.empty.
Type ls file*.
Type ls -lFG file*.
Type ls -hla file*.
Type rm file*.empty.
Type echo *.
Type echo *.*.
Type echo ~.

4.

Change to your home directory.
Create a batch of empty files by typing (note the capitals) touch FILE1.empty file2.empty FILE3.empty file4.empty FILE5.empty.
Type ls *.empty.
Type ls F*.empty.
Type ls f*.empty.
Type ls [Ff]*.empty.
Type ls [Ff].empty.
Type rm [Ff]*.empty.
Type ls *.empty.

5.

Change to your desktop.
Type for I in {1..500} ; do touch FILE$I.empty; done.
Look at the desktop. Take a moment to absorb what just happened.
Type ls FILE*.empty.
Type rm FILE*.empty.

6.

Type echo "a".
Type echo "b".
Type echo "c".
Type echo "a" > file1.file.
Type ls.
Type cat file1.file.
Type echo "b" >> file1.file.
Type ls.
Type cat file1.file.
Type echo "c" >> file1.file.
Type cat file1.file.
Type wc file1.file.
Type wc -l file1.file (that's a lowercase l as in, well, lowercase).
Type man wc and read a bit. Type q to exit man.
Type echo "d" > file1.file.
Type wc -l file1.file.
What happened?

Type cat file1.file.
Type rm file1.file.

7.

Type date.
Type NOW=`date` (those are backquotes).
Type echo NOW.
Type echo $NOW.
Type pwd.
Type CURRDIR=`pwd`.
Type echo CURRDIR.
Type echo $CURRDIR.
Type mkdir -p "$CURRDIR/banana".
Type B="$CURRDIR/banana".
Type echo $B.
Type cd $B.
Type cd ~.
Type pwd.
Type rmdir $B.

Initializing your repository

You will start by initializing your repository. Your repository will be hosted in the GitHub server, but you can create a local copy in your home directory (we will refer to this as your *local repository*). Go to https://github.com/join?source=header-home and sign up for a new free GitHub account using your UChicago email and using your CNet as your username. Click on “Start a project”, and you name your repository ***CAAP-CS***. The description should be your name and CNET, e.g.:

Jose Reyes / reyesj5

Initialize your repository with README file by clicking on that option and writing what the purpose of this repository is, in this case lab submissions for the CAAP course. Also add the GNU General Public License and create your new repository. At this point, you should install Git for Desktop here:

Linux: http://git-scm.com/download/linux.

Mac/Windows (Make sure it’s added to your path variables): https://desktop.github.com/

Now open your terminal or command prompt and you are going to do the following:

- Checkout a local copy of your repository
- Create a directory in your home directory for each hw assignment, lab, and the final project.

To download a copy of your repository to your computer, change to a directory of your choice, where you will store your repository and all your work for the course. In the terminal or command prompt type this: **$ git clone https://github.com/yourRepoAddress (or you can do: svn checkout https://github.com/yourRepoAddress optionalName).** This basically creates a directory that is the copy of your new repository on the GitHub server.

Now change to the repository directory and create folders named Lab1, Lab2, Lab3, and Lab4, as well as hw1, hw2, hw3, and final_project.

- Inside each folder, create a file called readme.txt with your full name and cnet.

Creating a commit

If you make changes to your repository, the way to store those changes (and the updated versions of the modified files) is by creating a *commit*.

Creating a commit is a two-step process. First, you have to indicate what files you want to include in your commit. Let's say we want to create a commit that only includes the updated readme.txt file from Lab1. We can specify this operation explicitly using the git add command from the terminal:
git add Lab1/test.txt

This command will not print any output if it is successful and it simply tells the server that you are creating new files and/or directories which are not fully added until you commit.

To create the commit, use the git commit command. This command will take all the files you added with git add and will bundle them into a commit:
git commit -m "Updated readme.txt"

The text after the -m is a short message that describes the changes you have made since your last commit. Common examples of commit messages might be "Finished lab 1" or "Implemented insert function for data struct".

If you forget the -m parameter, Git will think that you forgot to specify a commit message. It will graciously open up a default editor so that you can enter such a message. This can be useful if you want to enter a longer commit message (including multi-line messages). We will experiment with this later.

Once you run the above command, you will see something like the following output:

```
[master 3e39c15] Updated test.txt
1 file changed, 1 insertion(+), 1 deletion(-)
```

You've created a commit, but you're not done yet: you haven't uploaded it to the server yet. Forgetting this step is actually a very common pitfall, so don't forget to upload your changes. You must use the git push command for your changes to actually be uploaded to the Git server. *If you don't, the graders will not be able to see your code*. Simply run the following command from the Linux command-line:
git push

You should see something like this output:

```
Writing objects: 100% (3/3), 274 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/username/CAAP-CS.git
  4885f1c..3e39c15  master -> master
```

You can ignore most of those messages. The important thing is to not see any warnings or error messages. You can verify that our Git server correctly received your commit by going to GitHub online. The directories and files should now show the updated content (your name with your CNetID) If you wrote some code, and it doesn't show up in the GitHub server, make sure you didn't forget to add your files, create a commit, and push the most recent commit to the server.

git add revisited and git status

Make a Lab0 directory and make a test.txt file with your name and cnet and commit. Now make a README.txt file in the LAB0 directory and further change to test.txt: Add a line with the text CAAP - Introduction to Software Development.
So, at this point, we have a file we have already committed (test.txt) but where the *local* version is now out of sync with the version in the GitHub server. Furthermore, earlier we created a README.txt file. Is it a part of our repository? You can use the following command to ask Git for a summary of the files it is tracking:

```
git status
```

This command should output something like this:

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   test.txt

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

    README.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Notice that there are two types of files listed here:

- Changes not staged for commit: This is a list of files that Git knows about and have been modified since your last commit, but which have not been added to a commit (with git add). Note that we *did* use git add previously with test.txt (which is why Git is “tracking” that file), but we have not run git add since our last commit, which means the change we made to test.txt is not currently going to be included in any commit. Remember: committing is a two-step process (you git add the files that will be part of the commit, and then you create the commit).
- Untracked files: This is a list of files that Git has found in the same directory as your repository, but which Git isn’t keeping track of.

You may see some automatically generated files in your Untracked files section. Files that start with a pound sign (#) or end with a tilde should *not* be added to your repository. Files that end with a tilde are backup files created by some editors that are intended to help you restore your files if your computer crashes. In general, files that are automatically generated should not be committed to your repository. Other people should be able to generate their own versions, if necessary.

So, let’s go ahead and add README.txt:

git add README.txt

And re-run git status. You should see something like this:
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
new file: README.txt

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

Notice how there is now a new category of files: Changes to be committed. Adding README.txt not only added the file to your repository, it also staged it into the next commit (which, remember, won't happen until you actually run git commit).
If we now add test.txt:

git add test.txt

The output of git status should now look like this:
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

new file: README.txt
modified: test.txt

Now, we are going to create a commit with these changes. Notice how we are not going to use the -m parameter to git commit:
git commit

When you omit -m, Git will open a terminal text editor where you can write your commit message, including multiline commit messages. By default, the CS machines will use Vim for this; if you want to change your default command-line editor, add a line like this:
EDITOR=emacs
(I would recommend getting and using sublime for python and C, and Brackets for HTML and CSS)

At the end of the .bashrc file in your home directory.
Once Git opens your editor of choice, you will see something like this:

Please enter the commit message for your changes. Lines starting
with '#' will be ignored, and an empty message aborts the commit.
On branch master
Your branch is up-to-date with 'origin/master'.
#
Changes to be committed:
new file: README.txt

```
#       modified:   test.txt
#
```

Now, type in the following commit message:
Lab 1 updates:

```
- Added README.txt
- Updated test.txt file
```

Then, just save the file and exit (using the appropriate commands in your editor of choice). This will complete the commit, and you will see a message like this:

```
[master 9119c6f] Lab 1 updates
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 README.txt
```

Now, edit both files and add an extra line to each of them with the text Git is pretty cool. Running git status should now show the following:

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.

    (use "git push" to publish your local commits)

Changes not staged for commit:

    (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.txt
    modified:   test.txt
```

If we want to create a commit with these changes, we could simply run git add twice (once for each file) but, fortunately, we can also do this:

```
git add -u
```

This will add every file that Git is tracking, and will ignore untracked files. There are a few other shortcuts for adding multiple files, like git add . and git add --all, but we suggest you avoid them, since they can result in adding files you did not intend to add to your repository.
So, if you run git add -u and create a commit:

```
git commit -m "A few more changes"
```

git status will now show this:

```
On branch master
Your branch is ahead of 'origin/master' by 2 commits.

    (use "git push" to publish your local commits)
    nothing to commit, working directory clean
```

The message Your branch is ahead of 'origin/master' by 2 commits. is telling you that your local repository contains two commits that have not yet been uploaded to the GitHub server. In fact, if you go to GitHub you'll see that the two commits we just created are nowhere to be seen. As helpfully

pointed out by the above output, all we need to do is run git push, which should show something like this:

```
Counting objects: 8, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (8/8), 730 bytes | 0 bytes/s, done.
Total 8 (delta 1), reused 0 (delta 0)
To https://mit.cs.uchicago.edu/cmsc22000-spr-18/borja.git
        3e39c15..53462fb  master -> master
```

Go to GitHub again. Do you see the updates in your repository? Click on “Commits” (above the file listing in your repository). If you click on the individual commits, you will be able to see the exact changes that were included in each commit.
Now, git status will look like this:

```
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

If you see nothing to commit, working directory clean, that means that there are no changes in your local repository since the last commit you created (and, additionally, the above output also tells us that all our commits have also been uploaded to the GitHub server)

Working from multiple locations

So far, you have a local repository in your CS home directory, which you have been uploading to the GitHub server using the git push command. However, if you work from multiple locations (e.g., on a CS machine but also from your laptop), you will need to be able to create a local repository in those locations too. You can do this by running the git clone command.

This will create a local repository that “clones” the contents of the repository in the GitHub server. If you have a laptop with you (and have access to Git on that laptop), try running git clone there. If not, try creating a clone in a different directory on the CS machine you are on. For example:

```
mkdir /tmp/$USER/caap
cd /tmp/$USER/caap
git clone https://github.com/yourRepoAddress
```

Make sure to replace username with your Git username!
Now, in the local repository in your home directory, add a line to test.txt with the text One more change!. Create a commit for that change and push it to GitHub (you should know how to do this by now, but make sure to ask for help if you’re unsure of how to proceed).
Next, in the *second* local repository (the one you just created either on your laptop or on a separate location in the CS machine), check if that change appears in the test.txt. It will not, because you have not yet downloaded the latest commits from the repository. You can do this by running this:

```
git pull
```

This should output something like this:

```
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://mit.cs.uchicago.edu/cmsc22000-spr-18/borja
   53462fb..0c29617  master     -> origin/master
Updating 53462fb..0c29617
Fast-forward
 test.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
```

If you have multiple local repositories (e.g., one on a CS machine and one on your laptop), it is very important that you remember to run git pull before you start working, and that you git push any changes you make. Otherwise, your local repositories (and the repository on the GitHub server) may *diverge* leading to a messy situation called a *merge conflict* (we will be exploring these in a future Git lab). This will be specially important once you start using Git for its intended purpose: to collaborate with multiple developers, where each developer will have their own local repository, and it will become easier for some developers' code to diverge from others'.

Discarding changes and unstaging

One of the benefits of using a version control system is that it is very easy to inspect the history of changes to a given file, as well as to undo changes we did not intend to make. For example, edit test.txt to remove all its contents. git status will tell us this:

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

If we want to discard the changes we made to test.txt, all we have to do is follow the helpful advice provided by the above output:

```
git checkout -- test.txt
```

If you open test.txt, you'll see that its contents have been magically restored!
Now, edit test.txt and README.txt to add an additional line with the text Hopefully our last change....
Run git add -u but don't commit it just yet. git status will show this:

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
modified:   README.txt
modified:   test.txt
```

Now, let's say we realized we want to commit the changes to README.txt, but not to test.txt. However, we've already told git that we want to include test.txt in the commit. Fortunately, we can "un-include" it (or "unstage" it, in Git lingo) by running this:

```
git reset HEAD test.txt
```

Now, git status will show the following:

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README.txt

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

    modified:   test.txt
```

Go ahead and run git commit. The commit will now include only README.txt.

Looking at the commit log

Once you have made multiple commits, you can see these commits, their dates, commit messages, author, etc. by typing git log. This command will open a scrollable interface (using the up/down arrow keys) that you can get out of by hitting q. As we saw earlier, you can also see the history of commits through GitHub's web interface, but it is also useful to be able to access the commit log directly from the terminal, without having to open a browser.
Each commit will have a *commit hash* (usually referred to as the *commit SHA*) that looks something like this:

```
9119c6ffcebc2e3540d587180236aaf1222ee63c
```

This is a unique identifier that we can use to refer to that commit elsewhere. For example, choose any commit from the commit log and run the following:

```
git show COMMIT_SHA
```

Make sure to replace COMMIT_SHA with a commit SHA that appears in your commit log.
This will show you the changes that were included in that commit. The output of git show can be a bit hard to parse at first but the most important thing to take into account is that any line starting with a + denotes a line that was added, and any line starting with a - denotes a line that was removed.

Exercises

If you've completed all the steps described above, congratulations, you've already earned 30 points (out of 60) in this lab! Make sure you remember to git push so the grader can verify you completed all the above tasks. If you're still working on the above tasks, don't worry: nothing is due at the end of this lab. Please note that the above tasks will be graded as follows:

- **Task 1**: 10 points for the Updated test.txt commit
- **Task 2**: 5 points for the Directories and files commits
- **Task 3**: 5 points for the A few more changes commit
- **Task 4**: 5 points for the commit for testing that git pull works correctly
- **Task 5**: 5 points for the commit with README.txt (after unstaging test.txt)

Before next week's lab, you must also complete the following tasks. Some of them can be done just with what you have learned in today's lab, but most of them will require that you find the exact Git command (or series of Git commands) on your own. This is a very useful skill to develop: most software developers never take a course on Git or read a full book on Git before starting to use it; they learn the basics (like you did in this lab), and then rely on online documentation to fill the gaps.

So, for the following tasks, you are allowed to obtain the answers in any way you want **EXCEPT** by asking someone (other than the TA) to help you. This means you cannot ask for hints, solutions, pointers to documentation, etc. from *anyone* (classmates, roommates, friends, parents, etc.). Please note that you are welcome to take the answer verbatim from a website, online reference, online forum, etc. as long as you provide *attribution* (i.e., you need to tell us where you found the answer). Of course, you must also follow the instructions you find in those references to complete the task you've been given.

Pro tip: Sometimes, just Googling for "how do I…" will yield the answer or, at least, some solid leads.

Task 6 (5 points)

Add the following file tasks.txt to the Lab0 directory and the following lines of text:

```
Task 6
------
```

Create a commit for this change with commit message Adding Task 8 (yes, exactly that commit message) but make sure you *don't push it*.
Wait! What an embarrassing typo! Find out how you can edit the commit message of an existing commit (i.e., the solution is not to create a new commit; you have to find out how to edit the commit message of the commit you just created). Update the commit message to be "Adding Task 6".
Edit tasks.txt to explain how you updated the commit message (feel free to simply copy-paste the command you ran and its output). Make sure to explain how you found out the answer to this questions! (including citing any relevant sources). When you're done editing tasks.txt, make sure to add, commit and push your changes.

Task 7 (5 points)
Take a look at the following project on GitHub: https://github.com/junegunn/fzf. All you need to know about this project is that it provides a very handy tool called fzf that is run from the terminal, and which can take some number of command-line arguments.

Notice how the GitHub web interface has a number of similarities with the GitHub web interface: you can explore the files in the repository, past commits, etc. You should be able to figure out how to clone this repository on your machine. Then, find the exact commit where the authors of this project added a --no-mouse option to the fzf command (hint: commit messages will usually mention when a new feature is added, and this project is no exception). Take into account that you should be able to find this out using only Git commands (you do not need to build the project, run it, etc.). Include the commit SHA and commit message in tasks.txt, and explain how you located that commit.

Task 8 (5 points)
Edit README.txt and add any content to the file. Figure out how you can get Git to tell you the changes you've made to the file relative to the latest commit. Note that this is different from using git show, as we have not yet committed these changes. Once you have figured this out, and updated tasks.txt accordingly, undo these changes using git checkout.

Task 9 (5 points)
Create a file called mistake.txt with any content. Add, commit, and push it to your repository.

Actually, adding that file was a mistake (duh!). Figure out how to remove that file from your repository, while keeping a record of the fact that the file existed at some point. In other words, we are not asking you to *undo* the commit that created the file. We're asking you to create a commit that will remove the file. Explain in tasks.txt how you did this.

Note: The next task asks you to do something similar, and this task can technically be accomplished using the same (more general) mechanism in Task 12. For this task, you should find a command that specifically allows you to remove files.

Task 10 (10 points)
Edit README.txt to add the text This is a mistake. Add and commit (but do not push) this change. Edit the file again to add the test This is also a mistake. Add and commit (but do not push) this change.

Now, let's say we want to remove those two changes. We could, of course, just edit the file again, remove those lines, and add/commit the updated file (the commit could have a message like Reverting changes from commits A and B). However, if those two commits contained a large number of changes, removing those changes manually could get really messy. Fortunately, Git provides a command that will take one or more commits, and create a new commit with the opposite changes from those commits (effectively undoing those commits)
Note: You may encounter instructions online on how to "undo" a commit (in the sense of completely removing it from the commit log). This is not what we're asking you to do: you must find a command that specifically takes one or more commits, and undoes them by creating a new commit (thus preserving the record of those original commits).

Task 11 (2 Extra Credit points)

Go to the GitHub site, open your repository, and go to settings. Find the Collaborators tab and add me as a collaborator to your repository: reyesj5

This is how I will be able to grade your labs, so it's required of everyone.