Introduction to Git

Daniel Lin

Outline

What is version control?

- A system that keeps records of your changes (commit)
  - Who
  - When
  - What
  - Why
- Allows for collaborative development
- Allows you to revert any changes and go back to a previous state.
- Notifies conflict between different versions and allows the possibility of merging

What is Git?

Git is one of many version control systems.

- Users can keep entire code and history on their location machines
  - Can make any changes without internet access
  - Except pushing and pulling changes from a remote server
- It is widely used, both because it's easy to set up and the hosting site, GitHub

Key Concepts

- Snapshots
- Commit
- Repositories
- Push/Pull
- Branches

Snapshots

- The way git keeps track of your code history
- Record of what all your files look like at a given point in time
- **You** decide when to take a snapshot and of what files
- Have the ability to go back to visit any snapshot

Commit

- The act of creating a snapshot
- Essentially, a project is made up of a bunch of commits
- **Commits** contain these information
  - How the files changed from previous state
  - A reference to the commit that came before it
  - Comment from the person who made the commit
  - A hash code name
    - e.g. b05d7611c608d160619ddaae23a5d8b02ee9d1d6

Repositories

- A collection of all the files and the history of those files
  - Consists of all your commits
  - Place where all your work is stored
- Can live on a local machine or on a remote server (GitHub, ECCO, CISER)
- The act of copying a repository from a remote server is called cloning
- Cloning from a remote server allows teams to work together

Push/Pull

- **Push**
  - The process of uploading your local changes to the remote repository
- **Pull**
  - The process of downloading commits that don't exist on your machines from a remote repository

Branches

- All commits in git live on some branch
- The main branch in a project is called the **master** branch

What does a typical project look like?

- A bunch of commits linked together that live on some branch, contained in a repository
- You can make change to the master branch directly
- Or you can make a new branch pointing to a specific commit in the master branch
- Once you're done with your changes, you can merge the new branch back into master

Tutorial

- *git init*
- *git status*
- *git add*
- *git diff*
- *git commit*
- *gitignore*
- *git push*
- *git pull*
- *git clone*

Install git and Create GitHub account

Install git

- Windows
  - https://git-scm.com/download/win
- Mac
  - https://git-scm.com/download/mac
- Linux
  - Command: sudo apt-get install git

Create GitHub account

- Go to GitHub

Set Up

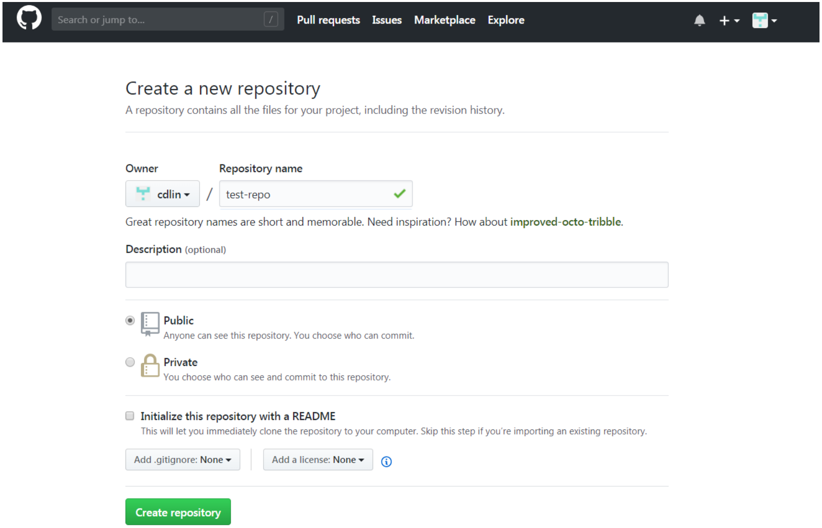
For your first time with Git on a new machine

```
$ git config --global user.name "Daniel Lin"
$ git config --global user.email "cdl77@cornell.edu"
$ git config --global color.ui "auto"
$ git config --global core.editor "nano"
```

You can check common git commands by typing

```
$ git help
```

Create an Online Repository



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: cdlin / Repository name: test-repo

Great repository names are short and memorable. Need inspiration? How about improved-octo-tribble.

Description (optional)

[✓ Public] Anyone can see this repository. You choose who can commit.

Private — You choose who can see and commit to this repository.

Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None

Add a license: None

Create a Local Repository

Let's create a "myrepo" folder on your desktop and change your directory to it

```
$ cd ~/Desktop
$ mkdir myrepo
$ cd myrepo/
```

Use *git init* to make it into a **repository**

```
$ git init
Initialized empty Git repository in
 C:/Users/Daniel/Desktop/myrepo/.git/
```

Add New File to Local Repo

You can use any text editor or run the *touch* command

```
$ touch code.do
$ ls
code.do
```

You can edit it using the *nano* command

```
$ nano code.do
```

Type a random sentence

```
This is a test.
```

Press CTRL+X to exit, press Y to save the change, press ENTER to confirm the file that is being overwritten

Git Status

Enter *git status*

```
$ git status
On branch master

No commits yet

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

                code.do

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

The output states that we are located on the master branch. The *Untracked files* message means that there are files in the directory that Git is not keeping track of.

Staging Environment

- A commit is a record of what files you have changed since the last time you made a commit.
- How do you tell git which files to put into a commit?
  - Add the files to the Staging Environment using *git add*
  - You can then tell git to package all the files in the staging environment into a commit using the *git commit* command

Git Add

We tell Git to stage untracked/modified files using *git add*

```
$ git add code.do
```

and check the status again

```
$ git status
On branch master

No commits yet

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

        new file:   code.do
```

So git knows that it needs to keep track of the file *code.do* but has not yet recorded the changes.

Git Commit

Let's *git commit* the added changes.

```
$ git commit -m "First draft code.do"
[master (root-commit) 887d395] First draft code.do
1 file changed, 1 insertion(+)
create mode 100644 code.do
```

- Git takes everything we have told it to save and stores a copy permanently inside the special .git directory.
- This commit has a short identifier 887d395.
- -m flag to record a comment that will help us remember later on what we did and why.

Now let's check the status again

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

Everything is up to date and committed.

gitignore

Sometimes you don't want git to track certain files (large datasets, logs, auxiliary files) Let's create some files

```
$ touch 2yr_paper.tex
$ touch 2yr_paper.aux
```

Git will ignore any changes to anything listed in *.gitignore*.

```
$ nano .gitignore
```

```
*.aux
*.log
*.gz
```

Press CTRL+X to exit, Y to save, ENTER to confirm

We have told Git to ignore any file whose name ends in .aux, .log and .gz.

gitignore

After we have created the .gitignore file, let's check *git status* again

```
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        2yr_paper.tex

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

Git no longer notices the auxiliary latex files.

Modifying files

Now let's edit our *code.do* file and check the status again.

```
$ nano code.do
```

```
This is a test.
Let's add a second line.
```

```
$ git status
On branch 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:   code.do

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

    .gitignore
    2yr_paper.tex

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

As well as the untracked files in the previous status check, Git also tells us that *code.do* has been modified.

Git diff

Use *git diff* to check the change from the previous commit before we add the files

```
$ git diff
diff --git a/code.do b/code.do
index 484ba93..067f664 100644
--- a/code.do
+++ b/code.do
@@ -1 +1,2 @@
This is a test.
+Let's add a second line.
```

Complete the Commit

Let's now add and commit these changes.

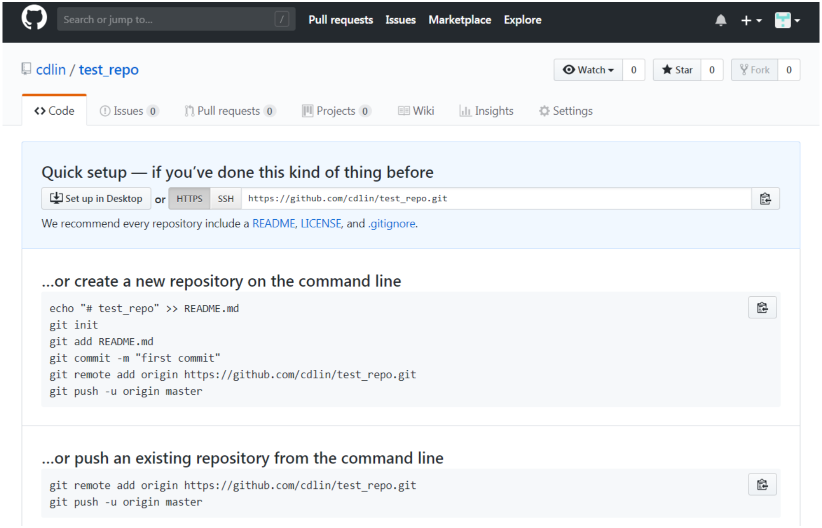
```
$ git add code.do
$ git status
$ git commit -m "New tex file and added line to code.do"
```

and checking the status....

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

... we're back to a clean sheet.

Connecting Local and Online Repositories



Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH https://github.com/cdlin/test_repo.git

We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# test_repo" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/cdlin/test_repo.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/cdlin/test_repo.git
git push -u origin master
```

Connecting Local and Online Repositories

Let's link our local repository with the GitHub repository we created earlier.
Check we're still in the same directory with *pwd*

```
$ pwd
/c/Users/Daniel/Desktop/myproject
$ git remote add origin https://github.com/cdlin/test_repo.git
```

This command tells git to add a remote named *origin* for the repository at *https://github.com/cdlin/test_repo.git*.

We can check that the command worked by typing

```
$ git remote -v
origin  https://github.com/cdlin/test_repo.git (fetch)
origin  https://github.com/cdlin/test_repo.git (push)
```

Git Push

We have linked our remote repository (GitHub) with our local repository. Let's push the commit from our local repository to the GitHub repository with *git push*

```
$ git push -u origin master
```

It may ask for your GitHub username and password

```
$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 234 bytes | 234.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/cdlin/test_repo.git
* [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Check your GitHub repository, everything we have done in your local repository should be there.

An overview of the Git Cycle

We should now be able to see the basic structure of how git works. Once a directory is set up as a Git repository using *git init*, any change that we make to that directory is noted by git. The workflow is then as follows:

1. Modify/create new folders and files.
2. Stage these changes ready to commit by using *git add*.
3. Commit the changes using *git commit* which permanently saves the current version of that repository.
4. Push the commit with *git push* to any remote repository

Undoing a “bad” commit

Suppose we have made some unwanted changes to a file and committed it. For example, we didn’t want the change we made to the *code.do*.

```
$ git log --oneline
57e91e5 (HEAD -> master, origin/master) New tex file and added line to code.do
887d395 First draft code.do
```

We want the version with the id 887d395. Use *git checkout*

```
$ git checkout 887d395 code.do
$ cat code.do
This is a test.
```

Note that we are not reversing the commit, but making a new one by replacing the code.do with a previous version.

```
$ git commit -m "Retrieved code.do from previous commit"
[master 9a664b0] Retrieved code.do from previous commit
1 file changed, 1 deletion(-)

$ git status
$ git push -u origin master
```

You can check your GitHub repository as well.

Hard delete unpublished commits

Now suppose that we really want to eliminate all traces of a series of particularly bad commits. For example, suppose we have committed the following:

```
$ nano code.do
```

Type a random sentence

```
This is a test.
This line will literally erase all my work.
```

```
$ git add code.do
$ git commit -m "Updated code"
[master 612ff94] Updated code
1 file changed, 1 insertion(+)
```

We can go back to our previous commit with *git reset*

```
$ git log --oneline
612ff94 (HEAD -> master) Updated code
9a664b0 (origin/master) Retrieved code.do from previous commit
57e91e5 New tex file and added line to code.do
887d395 First draft code.do

$ git reset --hard 9a664b0
HEAD is now at 9a664b0 Retrieved code.do from previous commit
```

Undo published commits with new commits

Note: you should never use *git reset* after the commits you want to undo have been pushed to a public repository. Instead, it is possible to use *git revert* to undo one or more target commits.

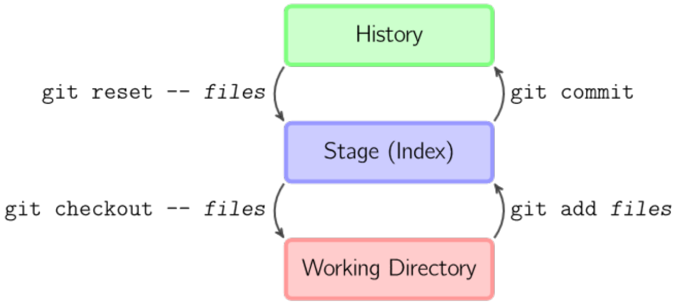
```
$ git log --oneline
9a664b0 (HEAD -> master, origin/master) Retrieved code.do from previous commit
57e91e5 New tex file and added line to code.do
887d395 First draft code.do

$ git revert 9a664b0
[master deba8ff] Revert "Retrieved code.do from previous commit"
1 file changed, 1 insertion(+)
```

This doesn't change the history of the commits and is therefore preferable.

Note: you can revert multiple commits at the same time; this will in turn create multiple new commits. See this blog post about all the differences between undoing commits.

Git Visualization



Git Pull

We can download changes from the remote repository (GitHub) to the local one with *git pull*:

```
$ git pull origin master
From https://github.com/cdlin/test_repo
* branch            master     -> FETCH_HEAD
Already up to date.
```

Pulling has no effect in this case because the two repositories are already synchronized. If someone else had pushed some changes to the repository on GitHub, though, this command would download them to our local repository.

Git Clone

Suppose you start working on a project with an existing GitHub repository. Instead of creating a new repository here with *git init*, we will *git clone* the existing repository from GitHub.

```
$cd ../myproject2
$ git clone https://github.com/cdlin/test_repo.git
Cloning into 'test_repo'...
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 9 (delta 0), reused 9 (delta 0), pack-reused 0
Unpacking objects: 100% (9/9), done.
```

We now has two copies of the repository on our Desktop.

Acknowledgment

This presentation follows the introduction to git by Meghan Nelson on Hubspot. Her tutorial can be found at:
https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners