Introduction to Git and Version Control

Lecture 1: Git ready!

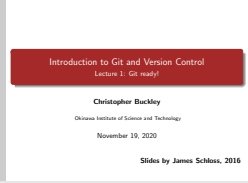
Christopher Buckley

Okinawa Institute of Science and Technology

November 19, 2020

OIST

Slides by James Schloss, 2016



Overview

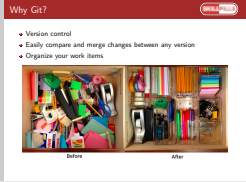
Overview
Why Git?
What is Git
Terminal Talk
Git basics
Local code
Nonlocal repos / github
Working alone

- Why Git? I'll tell you what my motivations are, but what are your motivations for being here?

Why Git?

- Version control
- Easily compare and merge changes between any version
- Organize your work items

Before | After



Imagine I asked you to remove the red sharpie marker from the left hand side?

- Difficulty finding it
- Could dive right in, but might get poked by lot of sharp things on the way in
- Or you could dump everything out and start all over

Why Git?

Why Git?

Imagine I asked you to remove this chair. What difficulties would you face?

- How to access it safely
- Can't remove it without fearing everything will fall

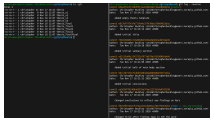
Traditional vs. Git Versioning

- What changed when
- Not limited to file name length to inform user of changes

```
christopher@christopher-ThinkPad-W541:~/git/nythesis$ ls -gtr
total 4
-rw-rw-r-- 1 christopher 51 Nov 17 18:22 thesis
-rw-rw-r-- 1 christopher  0 Nov 18 12:07 thesis_v1
-rw-rw-r-- 1 christopher  0 Nov 18 12:07 thesis_v2
-rw-rw-r-- 1 christopher  0 Nov 18 12:07 thesis_v3
-rw-rw-r-- 1 christopher  0 Nov 18 12:07 thesis_v4
-rw-rw-r-- 1 christopher  0 Nov 18 12:07 thesis_final
-rw-rw-r-- 1 christopher  0 Nov 18 12:07 thesis_final1
-rw-rw-r-- 1 christopher  0 Nov 18 12:07 thesis_final2
-rw-rw-r-- 1 christopher  0 Nov 18 12:07 thesis_final3
-rw-rw-r-- 1 christopher  0 Nov 18 12:07 thesis_finalfinal
christopher@christopher-ThinkPad-W541:~/git/nythesis$
```

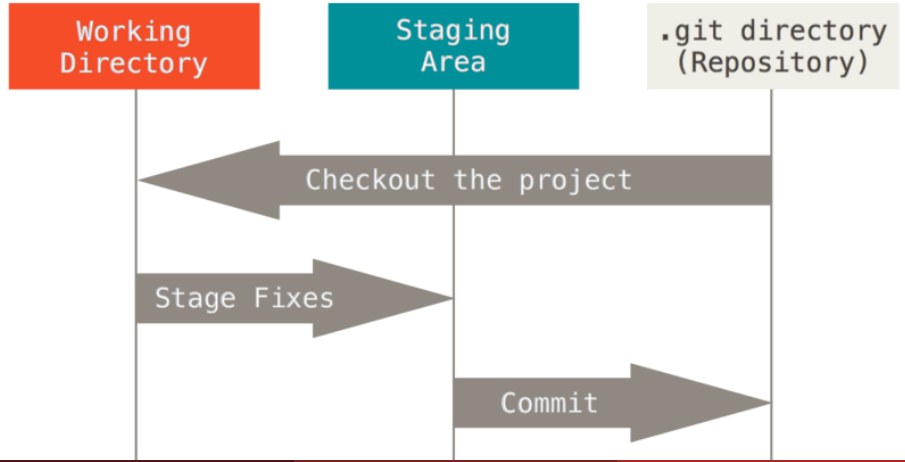
```
christopher@christopher-ThinkPad-W541:~/git/nythesis$ git log --reverse
commit 839a476257310dfd71ac8290cdefc64a60b8d944 (master)
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date:   Tue Nov 17 18:14:52 2020 +0900

    Added empty thesis template

commit e017bf79743fa7724d4c35f430e1e78064823e1a
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date:   Tue Nov 17 18:19:14 2020 +0900

    Added initial title

commit 750455880517cbdd5db25054e0e9815ed67da185
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date:   Tue Nov 17 18:20:38 2020 +0900

    Added initial summary section

commit a83135a08c1343723a7973a879e2431008b5a466
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date:   Tue Nov 17 18:21:09 2020 +0900

    Added initial bulk of main body section

commit 98c17b7472ef14bd75804d4ddb990de92f211ba
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date:   Tue Nov 17 18:21:31 2020 +0900

    Added initial conclusions

commit aa3034ffb24084d3f95e37ae222f265da87d3590
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date:   Tue Nov 17 18:22:03 2020 +0900

    Changed conclusions to reflect new findings on Mars

commit d918fbfe436cf474a34c6cde86ccf845f63e9cb4 (HEAD -> new_versioning)
Author: Christopher Buckley <15166572+topherbuckley@users.noreply.github.com>
Date:   Tue Nov 17 18:22:31 2020 +0900

    Changed title after finding typo in teh the word
```

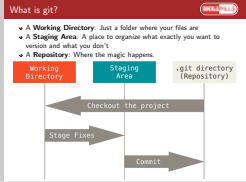

Traditional vs. Git Versioning
- What changed when
- Not limited to file name length to inform user of changes



What is git?

- A **Working Directory**: Just a folder where your files are
- A **Staging Area**: A place to organize what exactly you want to version and what you don't
- A **Repository**: Where the magic happens.



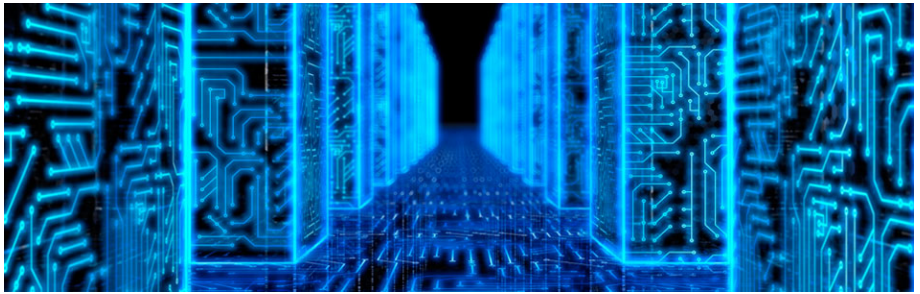


This chart might not make much sense now, but I hope it will before the end of these slides

Whether you realize it or not, you are already familiar with the "Working Directory". I'll save the staging area for a bit later, but first lets talk about what a repo is.

Repositories

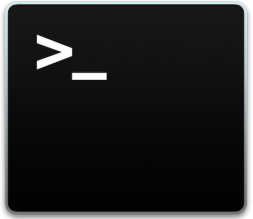
- A **repository** is a container for both your project data and all the items that allow interactions with git commands.
  - There are many sites to host your repository on (github, bitbucket), including your own local machine.
  - All of the essential parts of your repository can be found in the **.git** directory
  - GitHub (a website hosting Git repositories) $\neq$ Git (a set of tools for creating and managing those repositories).

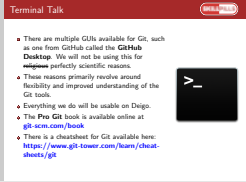


Repositories
A repository is a container for both your project data and all the items that allow interactions with git commands.
There are many sites to host your repository on (github, bitbucket), including your own local machine.
All of the essential parts of your repository can be found in the .git directory
GitHub (a website hosting Git repositories) ≠ Git (a set of tools for creating and managing those repositories).


Terminal Talk

- There are multiple GUIs available for Git, such as one from GitHub called the **GitHub Desktop**. We will not be using this for ~~religious~~ perfectly scientific reasons.
- These reasons primarily revolve around flexibility and improved understanding of the Git tools.
- Everything we do will be usable on Deigo.
- The **Pro Git** book is available online at **git-scm.com/book**
- There is a cheatsheet for Git available here: **https://www.git-tower.com/learn/cheat-sheets/git**





- I personally struggled with the terminal interface at first because most of the man pages use so much vocab I don't know to explain terms I don't know. Hopefully by the end of this mini-course you'll have the basic vocab down so you can help yourselves more efficiently going forward.

Create a Repo(sitory)

Let's **git** started.

- To initialize a git repository, simply type **git init** in a directory (preferably empty for now)
- This creates a folder **.git/**, where all your repository information is held.

Create a Repo(sitory)
Let's git started.
To initialize a git repository, simply type git init in a directory (preferably empty for now)
This creates a folder .git/, where all your repository information is held.
git

Quick Exercise

EXERCISE

1. Open a terminal
2. Create a new directory called **myFirstRepo** and enter it.
3. This is your Working Directory. Thats it!
4. Run **git init** in your Working Directory.
5. Take a peak in the newly created .git directory but don't touch anything quite yet.

Quick Exercise
EXERCISE
1. Open a terminal
2. Create a new directory called **myFirstRepo** and enter it.
3. This is your Working Directory. Thats it!
4. Run **git init** in your Working Directory.
5. Take a peak in the newly created .git directory but don't touch anything quite yet.

Commits

- Conceptually similar to "versions"
- The more effort you put into crafting these using the **staging area** the more helpful they are in the future.

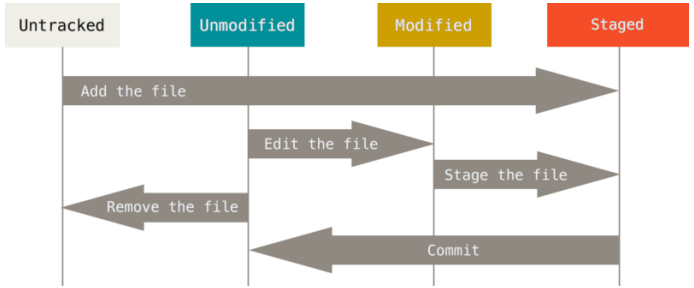
Commits
- Conceptually similar to "versions"
- The more effort you put into crafting these using the **staging area** the more helpful they are in the future.

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

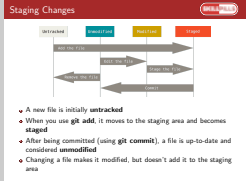
Before we talk about the Stage we need to understand what commits are.

Show lego photos now

Staging Changes



- A new file is initially **untracked**
- When you use **git add**, it moves to the staging area and becomes **staged**
- After being committed (using **git commit**), a file is up-to-date and considered **unmodified**
- Changing a file makes it modified, but doesn't add it to the staging area



Go through car repo example, e.g. adding wheels and doors, but committing only one or the other.

Currating the Stage before Committing

- Check what is on the stage with **git status**. Anything in green is staged.
- If you wish to unstage all changes, simply type **git reset**. This will remove everything from the stage, but keep your working directory untouched.
- **git reset** will work for individual files as well

```
git reset <file>
```

Currating the Stage before Committing
- Check what is on the stage with **git status**. Anything in green is staged.
- If you wish to unstage all changes, simply type **git reset**. This will remove everything from the stage, but keep your working directory untouched.
- **git reset** will work for individual files as well

```
git reset <file>
```

As you saw a few slides ago. Adding messy or unnecessary commits doesn't help anyone. Lets discuss how to make meaningful helpful commits using the stage. There are various ways to fix bad commits, but it takes a whole lot less work to do it right the first time.

Try out the Stage

EXERCISE

1. Create a new emtpy file **myfile.txt**
2. Check the status of everything with **git status**
3. Add **myfile.txt** to the stage vis **git add myfile.txt**
4. Check the status of everything again with **git status**. What changed?
5. Unstage the changes with **git reset myfile.txt**
6. Check the status of everything again with **git status**. What changed?

Try out the Stage
EXERCISE
Create a new emtpy file **myfile.txt**
Check the status of everything with **git status**
Add **myfile.txt** to the stage vis **git add myfile.txt**
Check the status of everything again with **git status**. What changed?
Unstage the changes with **git reset myfile.txt**
Check the status of everything again with **git status**. What changed?

Committing from the Stage

- Git keeps track of **commits**. Check these commits with **git log**. There's plenty of options to show only what you want or everything under the sun.
- **git status** checks any changes since the last commit.
- **git commit** commits everything in the *staging area* - git status shows these files in green by default.

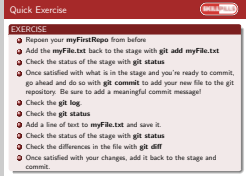
Committing from the Stage

- Git keeps track of **commits**. Check these commits with **git log**. There's plenty of options to show only what you want or everything under the sun.
- **git status** checks any changes since the last commit.
- **git commit** commits everything in the *staging area* - git status shows these files in green by default.

Quick Exercise

EXERCISE

1. Repoen your **myFirstRepo** from before
2. Add the **myFile.txt** back to the stage with **git add myFile.txt**
3. Check the status of the stage with **git status**
4. Once satisfied with what is in the stage and you're ready to commit, go ahead and do so with **git commit** to add your new file to the git repository. Be sure to add a meaningful commit message!
5. Check the **git log**.
6. Check the **git status**
7. Add a line of text to **myFile.txt** and save it.
8. Check the status of the stage with **git status**
9. Check the differences in the file with **git diff**
10. Once satisfied with your changes, add it back to the stage and commit.

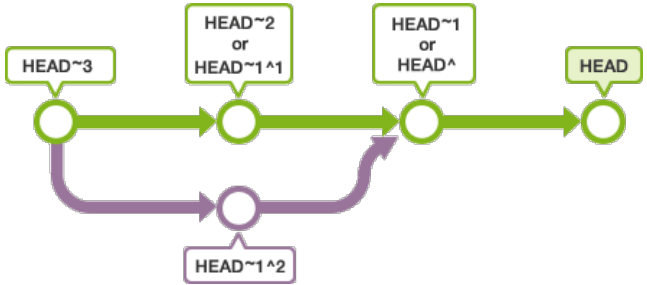


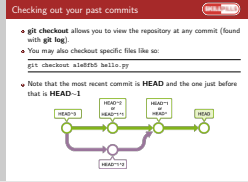
Checking out your past commits

- **git checkout** allows you to view the repository at any commit (found with **git log**).
- You may also checkout specific files like so:

```
git checkout a1e8fb5 hello.py
```

- Note that the most recent commit is **HEAD** and the one just before that is **HEAD∼1**

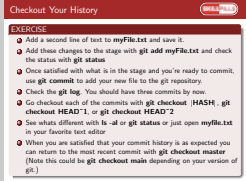




Checkout Your History

EXERCISE

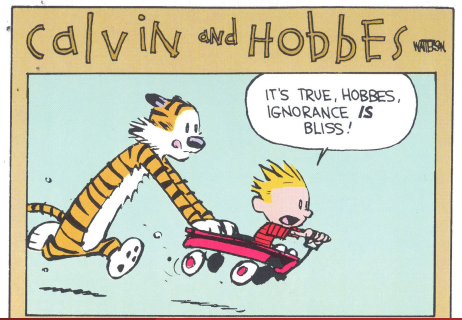
1. Add a second line of text to **myFile.txt** and save it.
2. Add these changes to the stage with **git add myFile.txt** and check the status with **git status**
3. Once satisfied with what is in the stage and you're ready to commit, use **git commit** to add your new file to the git repository.
4. Check the **git log**. You should have three commits by now.
5. Go checkout each of the commits with **git checkout** ⟨**HASH**⟩, **git checkout HEAD~1**, or **git checkout HEAD~2**
6. See whats different with **ls -al** or **git status** or just open **myfile.txt** in your favorite text editor
7. When you are satisfied that your commit history is as expected you can return to the most recent commit with **git checkout master** (Note this could be **git checkout main** depending on your version of git.)

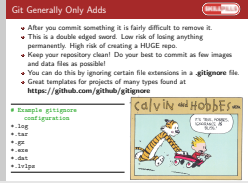


Git Generally Only Adds

- After you commit something it is fairly difficult to remove it.
- This is a double edged sword. Low risk of losing anything permanently. High risk of creating a HUGE repo.
- Keep your repository clean! Do your best to commit as few images and data files as possible!
- You can do this by ignoring certain file extensions in a **.gitignore** file.
- Great templates for projects of many types found at **https://github.com/github/gitignore**

```
# Example gitignore
    configuration
*.log
*.tar
*.gz
*.exe
*.dat
*.lvlps
```





My robot repo is now 500GB due to huge Solidworks CAD files.

Quick Exercise

EXERCISE

1. Touch multiple files with various extensions, one of which should be **.dat**.
2. Ignore the **.dat** file, but commit all the others.
3. Be sure to write a clear message describing what you did.
4. Check the **git log**

Quick Exercise
EXERCISE
1. Touch multiple files with various extensions, one of which should be .dat.
2. Ignore the .dat file, but commit all the others.
3. Be sure to write a clear message describing what you did.
4. Check the git log

git with it!

Now we move to the fun* stuff: working with **online repositories**.

- For this, we will be using **github**.
- We'll begin by creating a GitHub repository using the website.
  - If we're working on a project that's already hosted on a remote Git server, we can skip this step.
- Next, we use **git clone** to download a copy.
- From here, you can do the following:
  - **git push** to push any changes you may have to the online repository.
  - **git pull** to take any changes from the repository.

*Here, the word *fun* is subject to interpretation.

git with it!

Now we move to the fun* stuff: working with **online repositories**.

- For this, we will be using **github**.
- We'll begin by creating a GitHub repository using the website.
  - If we're working on a project that's already hosted on a remote Git server, we can skip this step.
- Next, we use **git clone** to download a copy.
- From here, you can do the following:
  - **git push** to push any changes you may have to the online repository.
  - **git pull** to take any changes from the repository.

*Here, the word *fun* is subject to interpretation.

Quick Exercise

EXERCISE

1. Fork the **https://github.com/oist/skillpill-git** repository using a browser.
2. Clone the forked repository* to your local disk:

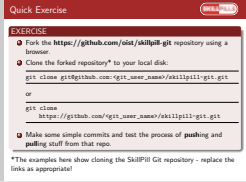
```
git clone git@github.com:<git_user_name>/skillpill-git.git
```

or

```
git clone
    https://github.com/<git_user_name>/skillpill-git.git
```

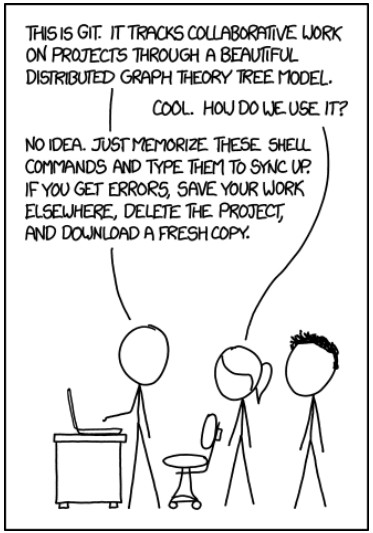
3. Make some simple commits and test the process of **push**ing and **pull**ing stuff from that repo.

*The examples here show cloning the SkillPill Git repository - replace the links as appropriate!



What it will feel like...

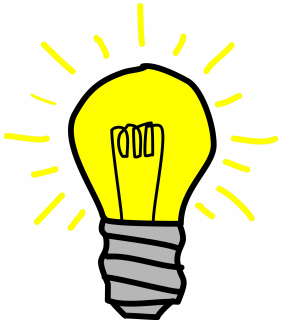
- git is not intuitive to start with, but it's a powerful tool for storing and restoring history, and working collaboratively with other people.
- The more you use it, the more you will like it. Think Stockholm syndrome.
- Operations that you use frequently will become easy.
- Operations you use infrequently, you can Google!



What it will feel like...
- git is not intuitive to start with, but it's a powerful tool for storing and restoring history, and working collaboratively with other people.
- The more you use it, the more you will like it. Think Stockholm syndrome.
- Operations that you use frequently will become easy.
- Operations you use infrequently, you can Google!

Final Comments

- git is weird. It's not intuitive, but it's the best way to collaborate with people on open projects.
- It's also great even if you don't collaborate!
- Whenever you are using git, think about other people and how they will perceive your comments. **Would you be able to understand your own cryptic commit messages?**
- You will make mistakes. Don't worry about it. Your entire history is backed up already. Learn from your mistakes and don't make them again!
- Read error messages carefully - they can be useful/informative/instructive.



Final Comments
git is weird. It's not intuitive, but it's the best way to collaborate with people on open projects.
It's also great even if you don't collaborate!
Whenever you are using git, think about other people and how they will perceive your comments. Would you be able to understand your own cryptic commit messages?
You will make mistakes. Don't worry about it. Your entire history is backed up already. Learn from your mistakes and don't make them again!
Read error messages carefully - they can be useful/informative/instructive.

(If Time Allows) Modifying Previous Commits

- If you commit something that turns out to be a mistake, don't worry! There are plenty of tools to rework commits.
- Some are more powerful (and potentially distructive than others)
- Non-destructive: (Leaves history intact)
  - **revert**
- Potentially destructive: (Changes history)
  - **rebase**
- Danger Zone: (Can erase history)
  - **reflog**

(If Time Allows) Modifying Previous Commits
- If you commit something that turns out to be a mistake, don't worry! There are plenty of tools to rework commits.
- Some are more powerful (and potentially distructive than others)
- Non-destructive: (Leaves history intact)
  - revert
- Potentially destructive: (Changes history)
  - rebase
- Danger Zone: (Can erase history)
  - reflog

Using Revert

- Revert makes a new commit showing that you reverted a previous commit.
- Pro: This is very useful for public repos (next session) where you want to show exactly what you've undone to others.
- Con: This can make your commit history quite messy if used too often.

EXERCISE

1. Make a few commits to your **myFirstRepo** or just use the existing history of your skill-pill fork
2. Find a commit you want to revert using **git log** and **git show** or **git diff**
3. Revert that commit with **git revert** ⟨**HASH**⟩ or **git revert HEAD~**⟨**#**⟩

Using Revert
- Revert makes a new commit showing that you reverted a previous commit.
- Pro: This is very useful for public repos (next session) where you want to show exactly what you've undone to others.
- Con: This can make your commit history quite messy if used too often.

EXERCISE
1. Make a few commits to your **myFirstRepo** or just use the existing history of your skill-pill fork
2. Find a commit you want to revert using **git log** and **git show** or **git diff**
3. Revert that commit with **git revert** ⟨**HASH**⟩ or **git revert HEAD~**⟨**#**⟩

Using Rebase

- Rebase rewrites the commit history by starting from specified base commit and choosing what to do with each commit all the way to the current HEAD.
- Pro: Great for removing WIP commits or otherwise meaningless commits. Use it to clean up your local history before pushing to a public repo.
- Con: This has the possiblity to create a lot of conflicts if used in a shared repo (as one person's history will differ from another). Generally rebase should not be used on any commits you have pushed to a public repo.

EXERCISE

1. Make a few commits to your **myFirstRepo** or just use the existing history of your skill-pill fork
2. Making a meaningless WIP commit
3. Use **git rebase -i** ⟨**HASH**⟩ and follow the instructions

Using Rebase
- Rebase rewrites the commit history by starting from specified base commit and choosing what to do with each commit all the way to the current HEAD.
- Pro: Great for removing WIP commits or otherwise meaningless commits. Use it to clean up your local history before pushing to a public repo.
- Con: This has the possiblity to create a lot of conflicts if used in a shared repo (as one person's history will differ from another). Generally rebase should not be used on any commits you have pushed to a public repo.

EXERCISE
1. Make a few commits to your **myFirstRepo** or just use the existing history of your skill-pill fork
2. Making a meaningless WIP commit
3. Use **git rebase -i** ⟨**HASH**⟩ and follow the instructions