

Git



Prerequisites

Command Line

What you'll learn

- What is Git?
- Git subcommands: help, init, status, add, rm, commit, config, log, reset, revert, checkout, clone, push, pull, diff
- Undoing Git mistakes
- GitHub

What is git?



What is git?

Git was invented in 2005 by Linus Torvalds, the same guy who created Linux. It was meant to be used as a tool to help developers collaborate on the Linux operating system project.

Git is a free, open source, distributed version control system. What does that mean?

Free means you don't have to pay for it And open-source means you can read the source code if you're curious about how it works.

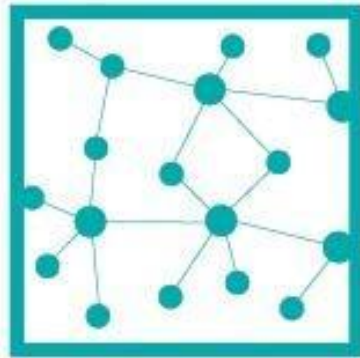
Distributed



What does distributed mean?

Most other version control systems are not distributed. They are centralized. That means they work like a library. The library stores all the books. You can check out some of the books and read them, but you have to return them to the library when you're done.

Distributed



Git is decentralized, meaning that there's no single authority. Everyone has a copy of everything. That means everyone has a copy of every library book. So if the library burns down, it doesn't matter. Because the books are widely distributed.

Being decentralized makes Git fast. If you already have a copy of every book, then it's very easy to checkout a single book. You don't have to get into your car and drive to the library. You just pull out your own copy.

Version Control

budget_estimation_final_v1.xlsx

budget_estimation_last_version_2.xlsx

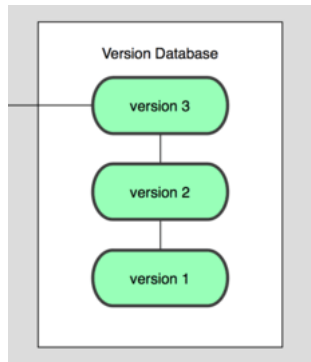
budget_estimation_May2014_new.xlsx

Version control is the solution to the problem demonstrated in this slide.

Say you have a spreadsheet named budget_estimation. Over time, you, and possibly many others, have updated this document, emailing the updated versions back-and-forth to each other. After a few weeks, you check your Documents folder and find three files named budget_estimation. One is named “final”, one is named “last version”, and one is named “May2014”. Which is the latest version? Is it the most recently modified version? Is it some combination of two different versions?

This is a file version control problem.

Version Control



A version control system keeps track of all the different versions of a file. It keeps them in order, from oldest to newest. And it allows you to avoid using some unreliable naming convention, like adding “new” or “latest” to the filename.

Backup Control



A version control system, besides allowing you to keep track of different versions of the same file, also allows you to store backups or a file. If you delete your copy of a file, you can ask Git to send you back the latest version of the file. If you somehow mess up the latest version of the file, you can ask Git to send you some previously saved version of the file. So a version control system not only keeps you organized, but it also prevents you from messing up. You can always recover.

git help

```
git --help
```

```
git --help add
```

```
git help add
```

Once installed, git is available as a command on the Command Line. Like gem, git has subcommands. You can view a list of some of the most common subcommands by using the --help flag.

If you'd like to see the man page for a specific command, you can type it after the help flag. Help itself is a subcommand, so you can also just type "git help" followed by the subcommand name.

Getting started

```
git init
```

```
ls -al
```

```
find .git
```

It's very easy to get started using git. Just pick a folder that you'd like to start tracking, cd to that folder in the Terminal, and type "git init".

All init does is create a directory inside the current directory named ".git". The .git directory contains all the storage and configuration for your git project (also known as a git repository or "repo").

git status

On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)

The git command you'll probably use most often is "status". This outputs a message letting you know the current status of your git repository. The status message contains 3 parts.

The first tells you what branch you're currently on. We'll talk about branches later, but every repo starts with one branch - the "master" branch.

The second line tells you information about your commit, which represents any items that are ready to get tracked and stored in git.

The third line is a useful hint letting you know what your next steps should be. In this case, since it's an empty folder, it's letting us know that the most common next step is to start adding files to the git repo.

git status

```
touch wyncode.txt
```

```
git status
```

Let's create an empty file named "wyncode.txt" using the touch command. Now our Git status has changed.

git status

Untracked files:

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

wyncode.txt

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

A new section was added to the status message. It's called "Untracked files:". Git is letting us know that the file we just created, "wyncode.txt", isn't being tracked by git. That means git has never seen it before, doesn't know anything about it, and doesn't have any previous versions of it stored away.

Like the helpful final message says, the next thing we probably will want to do is add that new file to git.

git add

```
git add wyncode.txt
```

```
git status
```

So let's use the add command to add this new file to git and see what happens to the status.

git add

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: wyncode.txt

The status updates to let us know that the file we just added, "wyncode.txt", is a new file and it's ready to be committed.

As a helpful note, the status let's us know what command to use to remove this file from Git, the "rm" command.

git rm

```
git rm --cached wyncode.txt  
git status
```

```
git add wyncode.txt  
git status
```

If we use the suggested “git rm” command, the status message goes right back to the “Untracked files” state. So let’s re-add the file again to get back to where we were, in the “Changes to be committed” state.

git commit

```
git commit -m "Add wyncode file"
```

Now that we have changes ready to be committed, let's run the "git commit" command.

Every commit must have a message associated with it. The message should briefly summarize what changes were made.

git config

Committer: Ed Toro <etoro@eds-macbook-pro.gateway.2wire.net>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

```
git config --global user.name "Your Name"  
git config --global user.email you@example.com
```

After doing this, you may fix the identity used for this commit with:

```
git commit --amend --reset-author
```

If you've never used git before, you should see a message letting you know that Git doesn't know your name and email address, but it needs those two items to complete your commit. So it just made a guess. The guess was probably incorrect. So, in another series of helpful messages, Git let's you know how to set your name and email address, and then shows you how to update the data in your last commit. So let's do that now.

git config

```
git config --global user.name "Ed Toro"  
git config --global user.email "ed@wyncode.co"  
  
git commit --amend --reset-author --no-edit
```

The “config” commands let’s me update how Git works. The global flag means that I’m updating Git for my whole laptop rather than the current repo.

The “commit” command Git provided me allows me to take my new name and email and apply it to the commit I just made. Note that I added the --no-edit flag because we’re not planning on updating the commit message, just the author.

git log

```
commit
60969bc312dfcfb108cfd10c8613f924c7a25f96
Author: Ed Toro <ed@wyncode.co>
Date:   Fri May 05 10:00:10 2014 -0400

    Add wyncode file
```

The “log” command outputs a list of all the commits Git is currently tracking. As you can see, there’s only one. It has an author (which includes my name and email address), a date, the commit message, and, up at the top, a unique identifier.

This means Git is currently tracking my last commit. If I’d like to see the list of files included in the commit, I can pass the `--stat` or `--name-only` flag to `git log`.

git config --global

```
cat ~/.gitconfig
```

```
[user]
```

```
name = Ed Toro
```

```
email = ed@wyncode.co
```

If you'd like to see where Git stores its global config, it's in a file named ".gitconfig" in your home directory.

Git Workflow

1. Create and edit files.
2. Add the files to the staging area.
3. Commit the staging area.

One of the things you may have noticed about that commit was that it was a 2-step process. First we created a file named “wyncode.txt”. Then we “git add”ed the file, which put it in a temporary staging area. Only then are we allowed to commit the change, which stores it permanently in Git.

What’s the point of the staging area? Why doesn’t the file just go straight into the repo?

git add

```
touch problem1.rb  
touch problem2.rb  
touch work_in_progress.rb  
  
git add '*.rb'
```

Let's backup and create a bunch more files, all empty Ruby files. Then let's "git add" them, but this time, instead of adding them one-at-a-time, let's use the Command Line wildcard asterisk to add all the files that end with .rb all at once.

git status

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: problem1.rb

new file: problem2.rb

new file: work_in_progress.rb

Now I have 3 files in my staging area. But work_in_progress.rb isn't ready to go yet. Only problem1 and problem2 are finished. So, following the advice Git provides, let's unstage work_in_progress.

git reset

```
git reset HEAD work_in_progress.rb
```

```
git status
```

“git reset HEAD” does the same thing as “git rm --cached” from earlier. But Git prefers that we use reset instead of rm in this case. Why? It’s a little complicated, but don’t worry about it. Just follow the advice Git provides.

git status

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: problem1.rb

new file: problem2.rb

Untracked files:

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

work_in_progress.rb

Now the git status has 2 sections. 1 section shows the files that are staged to be committed. The other shows untracked files. This is what we wanted. We don't want to track work_in_progress yet, only the 2 completed problems.

git commit (some files)

```
git commit -m "Solve problems 1 and 2"
```

```
git status
```

So let's commit the current staging area and check our status.

git status

Untracked files:

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

work_in_progress.rb

This is why Git supports a 2-stage commit process. It allows me to work on multiple things at the same time and only commit the stuff that's ready to go. Without a staging area, I'd have to make sure **all** my work was complete before every commit, which would be annoying. I'd have to somehow hide or delete my work in progress, commit my work, and then unhide or redo my work_in_progress.

git add ... oops

```
git add work_in_progress.rb
```

```
git commit -m "Committing some work"
```

```
git status
```

Let's say I did accidentally commit my work_in_progress.

git status

On branch master
nothing to commit, working directory clean

My git status tell me that I have nothing staged and my working directory is clean, meaning that I haven't created or updated any files. I'm perfect in sync with what Git is storing for me.

git log --name-only

commit 61da0be531d2137a4c3ad22aa76894c1ec540e74

Author: Ed Toro <ed@wyncode.co>

Date: Fri Apr 25 10:18:01 2014 -0400

Committing some work in progress.

work_in_progress.rb

In my Git log I can see the last bad commit I made. With the name-only flag, I can see the filename as well.

git rm

```
git rm work_in_progress.rb
```

```
git rm --cached work_in_progress.rb
```

I have a few options to undo this previous commit.

If I use “git rm” with the filename, then Git will remove both my copy of the file and stage the removal of it’s copy. But that’s not what I want. I want to keep my work_in_progress. I just want Git to stop tracking it.

If I used the cached flag, then Git will stage the removal of it’s version of the file, but keep mine. That’s what I want in this case.

git status

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

deleted: work_in_progress.rb

Untracked files:

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

work_in_progress.rb

"git status" let's me know that the staging area contains a task to delete the work_in_progress file. After I do that, Git won't know what work_in_progress is anymore, so it's now untracked.

Undo add file

```
git commit -m "Delete work_in_progress"
```

So I can commit that delete in the staging area, and my Git status will return back to what it was.

git add... opps (#2)

```
git add work_in_progress.rb  
git commit -m "Mistake #2"
```

Let's add the file again and try another way to back out of a mistake.

git log

commit 29edb23c18cf825426127267d8a15c92cae02ff4

Author: Ed Toro <ed@wyncode.co>

Date: Fri Apr 25 10:29:50 2014 -0400

Mistake #2

Once again I can see my 2nd mistake in the log.

git revert

```
git revert -n  
29edb23c18cf825426127267d8a15c92cae02ff4
```

```
git revert --abort  
git revert -n 29edb23
```

If, instead of removing the file, I want to just undo whatever I did in the last commit. In the last commit I added a file I didn't want to add. So the opposite of that is to remove the file.

git revert will do the opposite of whatever commit you specify. In this case I can specify the long id from the git log, telling git to revert this precise commit I did. With the "-n" flag, the result of this revert is staged. Without it, git would automatically commit the undo for me.

While Git uses these long id strings behind the scenes, it also realizes that, while useful to the computer, they're not very friendly to people. So Git will allow you to specify only the first few characters instead. If you specify enough characters for git to know what you mean, the shorter version will work. If by some chance two commits start with the same characters, you'll have to keep entering more characters until Git knows which one you're talking about.

So, since we staged this revert, we can abort it with the abort flag, and do it again using the shorter id. This worked for me.

git status

You are currently reverting commit 29edb23.
(all conflicts fixed: run "git revert --continue")
(use "git revert --abort" to cancel the revert operation)

Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

deleted: work_in_progress.rb

Before any commit, let's get into the habit of checking git status just to make sure everything is ok. This git status is a little bit different. It's letting you know that this commit is actually reverting another commit (which it's indicated with a shorter version of the commit id). It let's you know what to do if there's a conflict (which we'll talk about later) and what to do if you don't like what's happening (abort).

And, as usual, it shows you what actions are in the staging area. In this case, we're going to delete that work_in_progress file we didn't mean to add.

commit the revert

```
git commit -m "Revert mistake #2"
```

So let's commit this revert, since this is what we want. We want git to forget it ever saw this file.

oops

```
ls
```

```
problem1.rb  problem2.rb  wyncode.txt
```

Woops, now we have a problem. When git reverted the last commit, it not only deleted it from Git, it also deleted our local copy of the file. We just lost all of our work in progress! What are we going to do?

retrieve a backup

commit 87032f2843d4f9571ebf4078d0d9a936eae0fb33

Author: Ed Toro <ed@wyncode.co>

Date: Fri Apr 25 10:55:19 2014 -0400

Revert mistake #2

commit 29edb23c18cf825426127267d8a15c92cae02ff4

Author: Ed Toro <ed@wyncode.co>

Date: Fri Apr 25 10:29:50 2014 -0400

Mistake #2

Recall that git is not just a version control system. It's also a backup system. It keeps track of a copy of every file you've committed. Even if you delete the file, it only deletes it from the latest version. All the previous versions of the file still exist. So let's grab a copy of the version of the file we originally committed before the revert.

Let's pull up the git log and see what's going on. Each commit represents a version of our files that Git is tracking. In the "Mistake #2" commit, the file exists. In the "Revert mistake #2" commit, it doesn't anymore. So we need to travel back in time to "Mistake #2", grab the file we want, and bring it into our current workspace.

git checkout

```
git checkout  
29edb23c18cf825426127267d8a15c92cae  
02ff4 work_in_progress.rb
```

Like a library, git has a checkout subcommand. The checkout subcommand lets you pull files from any previous commit, even if they were later updated or destroyed. The format is the word “git”, followed by the subcommand “checkout”, followed by the id of the commit that represents the point in time you’d like to travel back to, followed by the name of the file you’d like to retrieve from that point in time.

git status

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: work_in_progress.rb

```
git reset HEAD work_in_progress.rb
```

And voila! Our missing file is back from the dead!

Git automatically staged the file for us. It assumes that, since we've pulled this file from the past, we probably want to do something with it right now. That's nice, but that's not what we want, so let's take Git's advice and unstage the file so we can continue working on it.

Git can get complicated. But the point of this exercise is that, once you start using it, it's actually hard to "mess up". If you accidentally do anything (delete a file, add a file, update a file), you can usually go back to a previous version (as long as git knows about it).

Do some damage

```
rm problem1.rb
```

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

```
deleted:    problem1.rb
```

Case in point, let's mess some stuff up. Let's delete an important file. Git status will now tell us that a file it's tracking has been deleted.

Do some damage

```
echo "Hello world" >> problem2.rb
```

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: problem1.rb

modified: problem2.rb

Let's also add some junk text to the end of one of our files. Now git status tells us that one file was deleted, another was modified. But it was all a mistake!

git reset

```
git reset --hard
```

Now we've got a bunch of bad edits and deletes in this directory. We can try to back out from them, one at a time, which is git's recommendation. Or we can just wave the white flag and tell git to just take us back to where we came from, before we made all these awful changes. Take us back to the last commit.

The "git reset" command will do just that. If you add the "--hard" flag, it'll find every file it knows about and *overwrite* it with the last version committed. In one command, all your local edits go away.

This is the kind of command you'd only want to use in an emergency. You're telling git that everything you've done since the last commit should be undone.

Git will never forget something that has been committed. But items that are untracked, that's on you. You have to make sure those files don't get deleted or reverted accidentally.

Git protects you only up to it's last commit. You can still lose any "work in progress" since then.

git status

Untracked files:

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

work_in_progress.rb

Speaking of work in progress, our work_in_progress.rb wasn't touched by the hard reset command. We told Git to forget about this file, so it won't touch it during a reset. It only resets files it knows about.

Git snapshots

```
rm work_in_progress.rb  
  
echo "puts 'Hello world'" >>  
problem1.rb
```

We've talked about adding files and removing files. Let's talk about editing files.

First, let's remove the untracked `work_in_progress` file so we can start with a clean status.

Then, let's append a line of Ruby code to the end of `problem1`.

git status

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: problem1.rb

As you would expect, Git lets you know that problem1 has been modified since the last commit.

Git snapshots

```
git add problem1.rb
```

```
git status
```

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: problem1.rb

Let's stage this modified file to get it ready to be committed. The status tells us now that the file is ready "to be committed".

Git snapshots

```
echo "puts 'Again'" >> problem1.rb
```

Now let's do something weird. Let's edit that file again by appending another line of Ruby code to the end. What do you think is going to happen to the "git status"?

Is the file still staged for a commit? Do my edits get included in the staging area?

Git snapshots

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: problem1.rb

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: problem1.rb

Check it out. The original version we added is still staged. The version we edited is unstaged.

The point of this example is that Git commits track **snapshots** of files. When you add a file to the stage, git takes a snapshot of the file at that point in time and adds that snapshot to the stage. You can continue working on the file (editing, deleting, whatever) and your snapshot won't change.

If you'd like your new edits to be part of your staged snapshot, you have to `add` them. If the new edits aren't ready yet, you can just commit the snapshot and keep working.

[Try to better explain the concept of a full-fidelity snapshot - like a 3-d picture of your home that's real, or copying files regularly.]

Where's GitHub?

```
find .git
```

```
.git//objects
.git//objects/02
.git//objects/02/f51b1d22e7f2565f46594c12409aa8c85a928f
.git//objects/14
.git//objects/14/d22b7af20ea558594e5146dc84c2abbc7a5a47
.git//objects/17
.git//objects/17/5de0931c19fa6ee1e9d4e24ef6c1950a31e946
.git//objects/18
.git//objects/18/bc60095e0c8c6585d7f95a17ad72b564104748
.git//objects/29
.git//objects/29/edb23c18cf825426127267d8a15c92cae02ff4
.git//objects/2b
```

We've seen adding files with `git add`, deleting files with `git rm`, editing files, and undoing mistakes. And we've managed to do all this without once mentioning GitHub.

Why?

Because Git and GitHub are separate things. You can use the Git software all by itself, on your laptop, to keep track of and backup different versions of files.

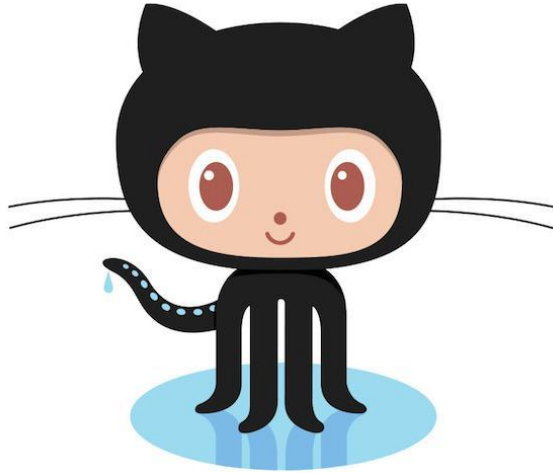
But that's not decentralized. You are the only library.

You can backup your Git repo yourself easily. Just backup the current directory and all subdirectories.

All the data being tracked by Git, including all previous versions of your files, are stored in the `.git` subdirectory. The file and directory names look weird, but I assure you the data is there.

So one way to achieve decentralization is to backup your whole laptop filesystem regularly, your Git repos included. There are a number of services available for that.

Where's GitHub?



But what if you'd like to collaborate with someone on a project? How do you share your repo with someone?

You can send your partner the entire directory, including the `.git` subdirectory, perhaps in a big zip file.

But how do you keep things in sync? What if he/she wants to make a change? How does your partner send changes back to you?

We still have a file versioning problem, only this time with a whole directory instead of just one file. Git wouldn't be very interesting if it didn't have a solution to this problem.


Git has a built-in feature to resolve this problem. You can run the Git program as a web server. That means other computers can connect to your computer, checking in periodically to make sure they have the latest versions of everything, and uploading their own changes. You and your partner are effectively sharing a single repo. You make commits locally. Your partner makes commits by pushing code to you over the internet.

But that's not an easy thing to set up.

That's where GitHub comes in. It's a cloud storage solution made for sharing and collaborating on Git repositories.


GitHub Repos


Owner **Repository name**

PUBLIC  eddroid /

Great repository names are short and memorable. Need inspiration? How about [turbo-octo-batman](#).

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 allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None** ▾ | Add a license: **None** ▾ ⓘ

Create repository

GitHub repos are easy to create. If you have a GitHub account, just click on the menu in the upper-right corner of the screen. Select a name for your repo. Select whether the repo should be public or private. Public repos are free. Private ones aren't.

Finally, you can allow GitHub to drop some basic files into your new repo automatically. Since we've already created the repo locally, we'll skip this step (just like GitHub recommends).

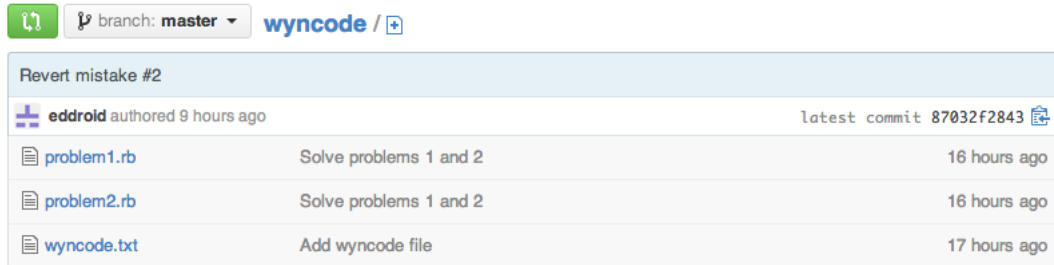
GitHub Repos

Push an existing repository from the command line

```
git remote add origin git@github.com:eddroid/wynocode.git  
git push -u origin master
```

Once I've created the repo in GitHub, there are nice clear instructions on how to link my local Git repo to this remote GitHub repo. So let's do that.

GitHub Repos



Once I finish doing what GitHub tells me to do, with I hit refresh on the repo page, my code appears.

Now I'm no longer the only library in town. Both I and GitHub now have a full-copy of my code. And we can talk to each other using git commands. And other people can talk to my GitHub repo as well. I'm now part of a decentralized network.

That means I get all the benefits of a decentralized network. If GitHub goes down, I can still work on my local copy until it recovers. If my laptop is destroyed, I can get a new one and resync my code with GitHub. If someone else downloads my code, then both me and GitHub can go up in flames and that third-party can continue using my code.

Up in flames

```
rm -rf {your_directory_name_here}
```

Let's try it out. Delete the entire directory you've been using so far. You'll have to use the `r` flag (for recursive) and the `f` flag (for force) to delete everything in the `“.git”` subdirectory.

git clone

```
git clone https://github.  
com/eddroid/wyncode.git
```

```
cd wyncode  
ls  
git log
```

Next, use the git clone command to download the repo you just created. My GitHub username is eddroid and the repo I created is named “wyncode”, so git clone will create a directory named wyncode and clone all of my files (and all previous versions of my files) into that directory.

When I cd into that new directory and ls, all of my files are there. If I check the git log, all of my previous commits are there too.

Collaboration

```
touch problem3.rb  
git add problem3.rb  
git commit -m "Solve problem 3"
```

GitHub is meant to help you collaborate with other coders. So let's pretend we're part of a team.

First, let's pretend to solve problem3 by creating an empty file named problem3.rb. Let's add and commit it.

Now, checking GitHub, I can hit refresh and see that my GitHub repo doesn't know about my solution to problem3 yet. I've told Git about my new file, but I still need to tell GitHub about the new file.

git status

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

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

Since I cloned my repo from GitHub, my local Git knows where it came from. It knows that my local Git repo is linked to a remote GitHub repo. So when I check my git status, it tells me exactly what I need to send my new commits up to GitHub: the git push command.

git push

Counting objects: 3, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (2/2), 229 bytes | 0 bytes/s, done.







Total 2 (delta 1), reused 0 (delta 0)

To <https://github.com/eddroid/wyncode.git>

87032f2..961257a master -> master

After asking me for my GitHub username and password, git will upload my commit to GitHub.

GitHub Repo Refresh

Solve problem 3		
 eddroid authored 9 minutes ago		latest commit 961257adf8 
 problem1.rb	Solve problems 1 and 2	17 hours ago
 problem2.rb	Solve problems 1 and 2	17 hours ago
 problem3.rb	Solve problem 3	9 minutes ago
 wyncode.txt	Add wyncode file	18 hours ago

When I hit refresh on the GitHub page for my repo, I can see my new file, problem3, along with the commit message I wrote. Now me and GitHub are all sync'd up.

I don't have to push after every commit. I can collect a bunch of commits. I don't even need internet access. I can just keep coding and committing. Then, when I'm ready, I can push up all my commits at once, with git push.

README

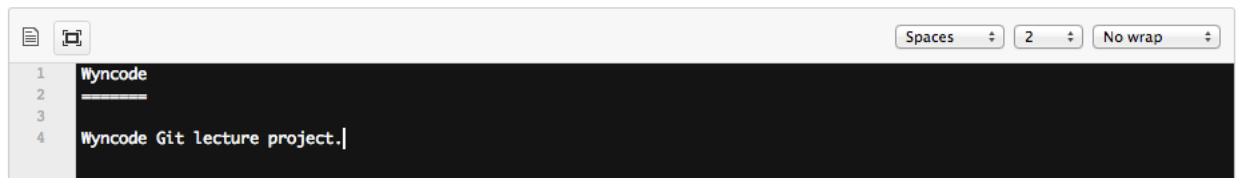
We recommend [adding a README](#) to this repository to help give people an overview of your project.

 **Add a README**

GitHub recommends that my repo have a file named README containing an overview of my project. It even provides a button that let's me create one on the site itself. So let's press it and see what happens.

README

wyncode / README.md or cancel



The screenshot shows a code editor interface. At the top, there's a header bar with the text "wyncode / README.md" and a button labeled "or cancel". Below this is a toolbar with icons for file operations and settings. The main editing area has a dark background and contains the following text:


```
1 wyncode
2
3
4 wyncode Git lecture project.|
```






On the right side of the editor, there are three dropdown menus: "Spaces" set to "2", and "No wrap".


Let's add a short message to the README using GitHub's web site and commit it using the form on the bottom.

README

Create README.md

 **eddroid** authored 4 minutes ago

 README.md	Create README.md
 problem1.rb	Solve problems 1 and 2
 problem2.rb	Solve problems 1 and 2
 problem3.rb	Solve problem 3
 wyncode.txt	Add wyncode file

 **README.md**

Wyncode

Wyncode Git lecture project.

GitHub displays the README file on my repo page as documentation. And it added a README.md file to my repo. But I don't have that file on my laptop. So now I'm the one who's out of sync.

git pull

```
From https://github.com/eddroid/wyncode
 961257a..c76778c master -> origin/master
Updating 961257a..c76778c
Fast-forward
 README.md | 4 +++++
 1 file changed, 4 insertions(+)
 create mode 100644 README.md
```

The “git pull” command downloads any commits in GitHub that I don’t have yet. The output is complex, but the gist is that I now have a new file named README.md, which I downloaded from the wyncode repo in my GitHub profile.

push and pull

`git push`
upload my commits to GitHub

`git pull`
download someone else's commits from
GitHub

I created a file that GitHub didn't have. Then I "git push"ed it to GitHub.

GitHub created a file that I didn't have. So I "git pull"ed it from GitHub.

Those are easy. But in a distributed system, keeping two repositories in sync isn't always so easy.

Imagine two libraries. If each one has a brand new book, they can send copies of those two books to each other.

But what if one library edits a book, and the other edits the same book differently. How does Git keep those conflicting changes in sync?

conflicting edits

```
echo "puts 'Hello world'" >> problem1.rb
```

```
git add problem1.rb
```

```
git commit -m "Update the answer to  
problem1"
```

First, let's add a line of Ruby code to the end of problem1.rb, add it, and commit it. But *don't* push it.

conflicting edits

branch: master wynocode / problem1.rb

edddroid 19 hours ago Solve problems 1 and 2

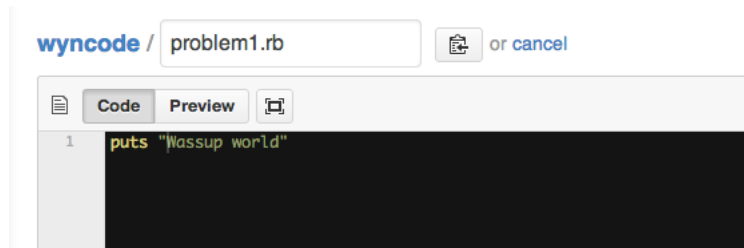
1 contributor

file 0 lines (0 sloc) 0.0 kb

Open Edit Raw Blame History Delete

In GitHub, let's click the "Edit" button and edit problem1.rb, but in a different way.

conflicting edits



On GitHub, let's make it say "Wassup world" instead of "Hello world". And let's use double-quotes instead of single-quotes.

Let's save our work in GitHub, which generates a new commit with our edits.

git push

```
To https://github.com/eddroid/wyncode.git
! [rejected]      master -> master (fetch first)
error: failed to push some refs to 'https://github.com/eddroid/wyncode.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
```

Now, from my laptop, when I try to push my commit to GitHub, the push is rejected. Git lets me know (in this very helpful hint) that I can't push up my commits until I pull down the latest commits. It recommends I do a "git pull" first. Ok.

git pull

```
From https://github.com/eddroid/wyncode
   c76778c..390b509  master    -> origin/master
Auto-merging problem1.rb
CONFLICT (content): Merge conflict in problem1.rb
Automatic merge failed; fix conflicts and then commit the
result.
```

Here's the output of my "git pull". Something went wrong. I edited problem1.rb, but so did someone else. Git attempted to merge GitHub's commit and my commit, but it didn't work. There's a conflict. I need to fix it before I can push up my commit.

This is the worst case scenario. Usually Git will be able to merge for you. If the file had multiple lines, and I edited the first line, and someone else edited the last line, Git would know what to do. But in this case we both edited the exact same line. And Git doesn't know which one to choose. So it's up to me to resolve the conflict.

cat problem1.rb

```
<<<<<<< HEAD
puts 'Hello world'
=====
puts "Wassup world"
>>>>>>> 390b509dfe4171ae4db23ed420409a671e1afd2b
```

When Git detects a conflict in a file, it edits the file by adding these extra lines. The code between HEAD and the equal signs is the way the line looks my Git. The code between the equal signs and the greater-than signs is what the code looks like in GitHub. I need to fix this. My choices are:

- 1) Keep the code that's already in GitHub.
- 2) Keep my code.
- 3) Manually merge my code and the GitHub code.

fix problem1.rb

```
echo 'puts "Hello world"' > problem1.rb
```

```
cat problem1.rb  
puts "Hello world"
```

At this point you should probably open a text editor, either SublimeText or Cloud9IDE, edit the file, and save your changes. But since this file is only 1 line, I'm just going to echo a line into it. I'm using 1 greater-than instead of 2, so I'm going to overwrite the file instead of append to it.

git status

You have unmerged paths.
(fix conflicts and run "git commit")

Unmerged paths:
(use "git add <file>..." to mark resolution)

both modified: problem1.rb

Now let's check the "git status" and see what git recommends I do next. "git status" says I have an unmerged conflict. Both my repo and the GitHub repo have edited problem1.rb. Once I've resolved this conflict, git says to "git add" the file. Let's do that.

git status

All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

Changes to be committed:

modified: problem1.rb

After adding the file like git recommended, the "git status" tells me that all the conflicts are fixed. All I have to do is a "git commit" to finish up. So let's do that.

resolve merge conflict

```
git commit -m "resolved merge  
conflict in problem1"
```

Don't forget to add a commit message to let people know what you've done.

git status

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

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

Now Git tells me I'm ahead of GitHub by 2 commits, my original edit to problem1.rb, and my commit to resolve the merge conflict in problem1.rb. Git recommends I do a "git push" now. Let's.

git push

Counting objects: 10, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (4/4), done.

Writing objects: 100% (6/6), 567 bytes | 0 bytes/s, done.

Total 6 (delta 2), reused 0 (delta 0)

To <https://github.com/eddroid/wyncode.git>

390b509..c29e0ce master -> master

And the “git push” is a success!

git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

Now my “git status” is clean. I’m sync’d up with GitHub.

git log

commit c29e0ce3fdcce4eafe858926bfccc3c0abf950e7

Merge: 422595b 390b509

resolve merge conflict in problem1

commit 390b509dfe4171ae4db23ed420409a671e1afd2b

Update the answer to problem1 on GitHub

commit 422595b5f76d1c2ce218f5eec7423475cceb8337

Update the answer to problem1

The git log output shows me that my commit came first, then the conflicting commit I did in GitHub came second, then my merge commit came last. The merge commit even lists the two commit IDs that I merged together.

And this is how you collaborate in GitHub. You make some commits and your team members make some commits. Whoever pushes to GitHub first wins. Everyone else has to pull first, before they can push. And if they pull down a merge conflict, they have to resolve that conflict before they can push.

One last thing...

```
echo "puts 'The end?'" > problem2.rb
```

One last thing before we finish up with Git for now. Let's edit problem2.rb by adding a single line of Ruby code.

git status

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: problem2.rb

The git status indicates that problem2 has been modified, but not staged.

git diff

```
diff --git a/problem2.rb b/problem2.rb
index e69de29..e60614c 100644
--- a/problem2.rb
+++ b/problem2.rb
@@ -0,0 +1 @@
+puts 'The end?'
```

Now let's say, at this point, I go out to lunch. Or maybe I leave for the day. Or maybe I leave early for the weekend. Or maybe some manager or client interrupts me. Whatever it is, I get distracted for a little while. When I get back, I see in my "git status" that I edited problem2. But I completely forgot what I changed. What was I working on?

"git status" only tells me *that* a file was modified. "git diff" tells me *how* it was modified. There's a lot of output here, but focus on the last line. The + sign means that I added a line. Everything after the + is what I added. So the diff is telling me that I updated problem2.rb, and I did so by adding a line of code containing "puts 'The end?'". Great, now I remember what I was working on!

The end?

<https://octodex.github.com/>



Git is definitely complicated. It can do a lot. We've gone through a ton of slides and we still haven't touched on all of it. There's still a whole other lecture on advanced git topics like branching, merging, and cherry-picking. Until then, entertain yourself with the octodex, a list of alternative versions of GitHub's Octocat logo. Because everyone loves the Octocat, even if he technically doesn't have 8 arms.