# Git

September 12, 2018

## 1 Git

### 1.1 Some Prerequisites

Before we start, make sure you have **Git** and **Atom** installed.

If you are on Mac, you will most likely have it already installed. Run the following command on terminal to check:

```
git --version
```

If you are on Windows, see: https://gitforwindows.org/

If you are on Linux, see: https://git-scm.com/download/linux

Next, install **Atom**: https://atom.io/

## 2 What is Git?

If you've used Google Docs before, you're already familiar with a version control system for editing files. In Google Docs, you can go back to the history of versions for a document, and easily restore a prior version if you make a lot of unwanted changes. Google Docs even tells you who edited which portions of each version, so you can contact the right people for large documents.

Git is also a version control program, but is much older and more versatile. Whereas Google Docs stores version history for specific documents, Git stores version history for entire projects (known as Git repositories ). Unlike Google Docs, Git allows you to make branches, or named working versions of a project, and you can quickly switch between them, remotely or locally. This branching feature is illustrated by the Git logo:

### 2.1 A quick example

To give you a more concrete idea of what branching is used for, here's an illustrative example.

Suppose you and your friend are working on a state-of-the-art Kaggle model. You have been working with your friend to make a basic model, storing it in the default Git branch, `master`. Now, you and your friend have a couple of different ideas on how to improve it further. You both want to start working on making a better model, and then compare which prototype performs better, but you don't want to get in the way of each other's work. So, you make a branch named `xg_boost`, and your friend makes a branch named `clockwork_rnn`. That way, both of you can independently work off of the same base code, without overwriting or affecting each other's branches.

If your model proves to be superior, you can merge your `xg_boost` branch into the `master` branch, and you can both work off of your new model. Alternatively, if your friend's model

performs better, they can merge their `clockwork_rnn` branch into the `master` branch and you and your friend can work on that new model together. As one last possibility, suppose neither the `xg_boost` model nor the `clockwork_rnn` model outperform the model in the master branch. Then, you can both easily switch back to the `master` branch with a single command, and try new ideas without skipping a beat. This is the value of Git, especially for complex or group projects.

## 3 Git vs GitHub

`git` is the command-line program that allows you to make use of the Git version control system. However, Git operates by default locally - that is, just offline on your computer. In order to actually share your code with others, you have to use a service like Github or Bitbucket. They both work very similarily, and basically allow you to upload your code to an online repository in the cloud.

The standard workflow is to work on your code locally, using `git` for version control, but when you want to push the latest functioning version of your code, you can upload it to the cloud maintained by Github/Bitbucket. Then, even friends who aren't located nearby geographically can pull your online code to their local computers, and work on your project remotely, pushing their own updates when needed.

## 4 Installing Git

You can install Git by visiting this link to Downloads of the official Git Website

## 5 Using Git Locally

We are now going to dive into how to use `git`! Don't worry, there are just a few basic commands, and once you get used to the workflow, it will feel painless and natural.

### 5.1 `git init`

First, we're going to learn how to actually make a Git repository. A repository (aka repo) is essentially a folder/directory that you specially indicate as being attached to a version control system. First, let's make a demo folder and copy some files into it:

```
cd ~
mkdir git-workshop && cd git-workshop
echo "Hello World" > hello.txt
```

To initialize a directory as a repo, navigate to the folder in question and run:

```
git init
```

### 5.2 `git status`, `git add` and `git commit`

Now, let's record our files into the version control system.

```
git status
```

This command should show that the file `hello.txt` is untracked. In order to have Git control this file, we use `git add`

`git add` will tell Git to track your files, or prime them to be "stamped" as a major step in development later in the `git commit` step. You usually just `add` the files that have changed since the last `commit`, or just the files that you want to track. Alternatively, you can `add` all the files that are in your repo with the `-A` option argument. Let's add all our files to be tracked.

```
git add -A
```

`git commit` is then used to make a named record of your work so far. All commits require a message to be attached, to describe what's changed since the last version. If you simply run `git commit`, Git will automatically open up your default command-line text editor to force you to enter a message, which can be a pain to work with. For quick messages, you can use the `-m` option command to give a message as a string. Let's commit our newly `add`ed files:

```
git commit -m "My first commit"
```

As you can see, we've successfully tracked and committed the file that we added to the empty repo.

### 5.3 `git branch` and `git merge`

Now, we're going to give a brief illustration of how to use Git's branching mechanism. Let's make a new branch, edit a file, and then merge that branch back into the `master` branch.

To open a new Git branch, use `git checkout -b <new_branch_name>`. Let's make a branch called `dev`.

```
cd ~/git-workshop
git checkout -b dev
```

If you run `git branch` without any arguments, Git will display all the branches currently associated with this repo. What are all the branches in our case?

```
* dev
master
```

Notice the `*` next to the `dev` branch - this indicates we are now on the `dev` branch, to switch back to `master`, we use `git checkout master`

Now, we're ready to make some changes without it affecting our `master` branch. Let's rewrite the contents of `hello.txt`.

```
echo "Hello, Berkeley" > hello.txt
```

Let's `add` and `commit` our changes. A quick shortcut to do this is the `-a` option argument of `git commit`, which automatically adds all changed files, and then commits them.

```
git commit -am "changed hello" <– "-am" is the same as doing "-a" and "-m"
```

If we `cat` (display the contents of) the `hello.txt` file in our `dev`branch, we see that it has in fact be overwritten:

`cat hello.txt`

However, if we switch back to the `master` branch, we find that our old version of `hello.txt` is still available once we're on that branch.

```
git checkout master
cat hello.txt
```

Suppose we're satisfied with our work in the `dev`, and now want to officiate it by merging it back with the `master` branch. To do this, we use the `git merge` command. The syntax `git merge <branch>` will merge the `<branch>` branch with the branch you are currently on. Right now, we're on the master branch (Quiz: how do we know that we are currently on `master`? How can we double-check which branch we are on?), so we run the following command to merge our `dev` branch with the `master` branch:

```
git merge dev
```

(Notice the `Fast-forward` description to this merge - there are several types of merges Git uses, and this one is the most basic - we simply fast-forward to the changes of the `dev` branch. You can learn about the other types of merges in the section below.)

Normally in our workflow, after we have merged the contents of our branch to `master` we can delete the branch: `git branch -d dev`

# 6 Using Git Remotely

Now that we've covered the basics of how Git works locally, let's make use of it's cloud-interfacing abilities. First, let's `push` our local `git-workshop` `master` branch to Github. We must first make an empty Github remote repository on the Github website, then `push` our local repo onto it.

## 6.1 `git push`

`git push` has a very simple syntax for pushing local repositories to remote repositories - `git push <remote_repo_url> <local_branch_name>` will push the `<local_branch_name>` branch into the remote repository at `<remote_repo_url>`. (By default, if no `<local_branch_name>` is given, the default is the current branch) Let's push our `git-workshops`'s `master` branch into the remote `git-demo` repository:

First, we must tell Git to push commits to our repository on GitHub. In order to do so, set up a **remote** as follows:

We can call the remote repository whatever we like; normally we refer to it as `origin`

```
git remote add origin {repository URL}
```

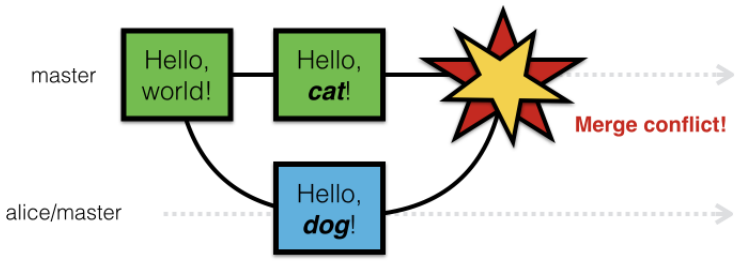
Now you can run:

```
git push origin master
```

## 6.2 `git pull`

Finally, we're going to cover the other command for working with remote repositories, `git pull`. To `fetch` and automatically `merge` any changes in the remote repo that aren't yet in your local repo, use the syntax `git pull <remote_repo_url> <remote_branch_name>`. (Again, by default if no `<remote_branch_name>` is specified, Git will try to guess which branch is associated with your current branch). Let's `pull` from our remote repository in case any changes have been made to it that we aren't aware of locally.

Let's simulate a second developer "pushing" to our branch by creating a file on github




```
cd ~/git-workshop
git pull origin master
```

Now when we `pull` from `origin` we should see the new file in our local directory

### 6.3 `git clone`

An alternative to making a local repo and then pulling remote files to it is to `clone` a remote repo directly as a new local repo on your computer. The `git clone` syntax is as follows: `git clone <remote_repo_url> <local_repo_name>`. (By default, if no `<local_repo_name>` is specified, Git will give the local repo the same name as the remote repo).

To `clone` all of the SUSA crash-course tutorials (including this one!) to your computer, run the following command in whichever directory you want to store the repo in:

```
cd ~
mkdir susa-crash-course && cd susa-crash-course
git clone https://github.com/SUSA-org/crash-course.git
```

## 7 Merge Conflicts

Scenario: you are working with a friend on a repository (on GitHub). You update a file on your branch, `dev`, and try to merge it into `master`, but your friend already updated the file there!

Git now has two commits (versions) of your files: one that you are trying to update and push to `master`, and one that was already pushed to `master` when you were working! This is called a **merge conflict**.

Let's create this scenario:

Simulate someone else making a change to the `hello.txt` through Github and commit those changes

Now let's change `hello.txt` in a different way locally and commit these changes

When you try to push, you should see:

```
Updates were rejected because the remote contains work that you do
not have locally
```

To fix this, we must `git pull origin master` first

Now you should see:

```
Auto-merging hello.txt
CONFLICT (content): Merge conflict in hello.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Anything between `HEAD` and the equal signs is what you currently have and below is what you are comparing with, to fix the merge conflict, you keep whichever line or whatever variation of the line that is correxct and then you delete everything else created

**Atom** has very good support for merge conflicts. Open your `git-workshop` repository in Atom, manually or by using the following command:

```
cd ~/git-workshop
atom .
```

After you're done fixing the merge conflict, run:

```
git commit -am "fixed merge"
git push origin master
```

### 7.1 `.gitignore`

Sometimes we do not want to track certain files such as log files or default files created by IDEs, to ignore these types of files, we create a .gitignore file:

```
touch test.log
```

We have created a log file we do not want to track, if we run `git status`, we see that git marks it as untracked

```
echo "*.log" > .gitignore
```

Here we create a .gitignore file, and tell it to ignore all files ending with `.log`, now we must `add` and `commit` this file

```
git add .gitignore
git commit -m "adding git ignore"
```

### 7.2 `git reset`

Say we make a mistake and want to revert our changes to a previous commit.

```
echo "Hello, Stanford" > hello.txt
```

if we haven't added and committed these changes yet we can simply type:

`git checkout -- hello.txt` ( can use . to revert everything in directory)

cat the contents of `hello.txt` to see that it is back to normal

Lets consider the case where we have accidently added and committed the file, use `git log` to see the unique identifier of the commit you want to revert to and then use `git reset --hard` to go back.

```
git log
git reset --hard "commit_id"
```

Be careful, with `git reset --hard` because it is a destructive command and you can not go back.

## 8 Additional Readings

For more information on the command-line shell, visit the Linux Command Line Guide

For more information on how to use Git, visit the official Pro Git Book

There is also a Git quick-guide cheatsheet, available here