Git on the commandline

In this section you will:

- install and configure Git locally
- create your own local clone of a repository
- create a new Git branch
- edit a file and stage your changes
- commit your changes
- push your changes to GitHub
- make a pull request
- merge upstream changes into your fork
- merge changes on GitHub into your local clone

So far we've done all our Git work using the GitHub website, but that's usually not the most appropriate way to work.

You'll find that most of your Git-related operations can and need to be done on the commandline.

Install/set up Git

```
sudo apt-get install git # for Debian/Ubuntu users
brew install git # for Mac OS X users with Homebrew installed
```

There are other ways of installing Git; you can even get a graphical Git application, that will include the commandline tools. These are described at:

http://git-scm.com/book/en/Getting-Started-Installing-Git

Tell Git who you are

First, you need to tell Git who you are:

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

Give GitHub your public keys

This is a great timesaver: if GitHub has your public keys, you can do all kinds of things from your commandline without needing to enter your GitHub password.

- https://github.com/settings/ssh

https://help.github.com/articles/generating-ssh-keys explains much better than I can how to generate a public key.

This tutorial assumes you have now added your public key to your GitHub account. If you haven't, you'll have to use *https* instead, and translate from the format of GitHub's *ssh* URLS.

For example, when you see:

```
git@github.com:evildmp/afraid-to-commit.git
```

you will instead need to use:

```
https://github.com/evildmp/afraid-to-commit.git
```

See https://gist.github.com/grawity/4392747 for a discussion of the different protocols.

Some basic Git operations

When we worked on GitHub, the basic work cycle was *fork* > *edit* > *commit* > *pull request* > *merge*. The same cycle, with a few differences, is what we will work through on the commandline.

Clone a repository

When you made a copy of the *Don't be afraid to commit* repository on GitHub, that was a *fork*. Getting a copy of a repository onto your local machine is called *cloning*. Copy the *ssh URL* from `https://github.com/<your github account>/afraid-to-commit`, then:

```
git clone git@github.com:<your github account>/afraid-to-commit.git
```

Change into the newly-created `afraid-to-commit` directory, where you'll find all the source code of the *Don't be afraid to commit* project.

Now you're in the **working directory**, the set of files that you currently have in front of you, available to edit. We want to know its status:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Create a new branch

Just as you did on GitHub, once again you're going to create a new branch, based on *master*, for new work to go into:

```
$ git checkout -b amend-my-name
Switched to a new branch 'amend-my-name'
```

`git checkout` is a command you'll use a lot, to switch between branches. The `-b` flag tells it to **create a new branch** at the same time. By default, the new branch is based upon whatever branch you were on.

You can also choose what to base the new branch on. A quite common thing to do is, just for example:

```
git checkout -b new-branch existing-branch
```

This creates a new branch `new-branch`, based on `existing-branch`.

Edit a file

1. find the `attendees_and_learners.rst` file in your working directory
2. after your name and email address, add your Github account name
3. save the file

`git status` is always useful:

```
$ git status
# On branch amend-my-name
# 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:   attendees_and_learners.rst
#
no changes added to commit (use "git add" and/or "git commit -a")
```

What this is telling us:

- we're on the *amend-my-name* branch
- that we have one modified file
- that there's nothing to commit

These changes will only be applied to this branch when they're committed. You can `git add` changed files, but until you commit they won't belong to any particular branch.

Note

When to branch

> You didn't actually *need* to create your new *amend-my-name* branch until you decided to commit. But creating your new branches before you start making changes makes it less likely that you will forget later, and commit things to the wrong branch.

Stage your changes

Git has a **staging area**, for files that you want to commit. On GitHub when you edit a file, you commit it as soon as you save it. On your machine, you can edit a number of files and commit them altogether.

Staging a file in Git's terminology means adding it to the staging area, in preparation for a commit.

Add your amended file to the staging area:

```
git add attendees_and_learners.rst
```

and check the result:

```
$ git status
# On branch amend-my-name
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   attendees_and_learners.rst
#
```

If there are other files you want to change, you can add them when you're ready; until you commit, they'll all be together in the staging area.

What gets staged?

It's not your files, but the **changes to your files**, that are staged. Make some further change to `attendees_and_learners.rst`, and run `git status`:

```
$ git status
# On branch amend-my-name
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   attendees_and_learners.rst
#
# 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:   attendees_and_learners.rst
#
```

Some of the changes in `attendees_and_learners.rst` will be committed, and the more recent ones will not.

- run `git add` on the file again to stage the newer changes

Commit your changes

When you're happy with your files, and have added the changes you want to commit to the staging area:

```
git commit -m "added my github name"
```

The `-m` flag is for the message ("added my github name") on the commit - every commit needs a commit message.

Push your changes to GitHub

When you made a change on GitHub, it not only saved the change and committed the file at the same time, it also showed up right away in your GitHub repository. Here there is an extra step: we need to **push** the files to GitHub.

If you were pushing changes from *master* locally to *master* on GitHub, you could just issue the command `git push` and let Git work out what needs to go where.

It's always better to be explicit though. What's more, you have multiple branches here, so you need to tell git *where* to push (i.e. back to the remote repository you cloned from, on GitHub) and *what* exactly to push (your new branch).

The repository you cloned from - yours - can be referred to as **origin**. The new branch is called *amend-my-name*. So:

```
$ git push origin amend-my-name
Counting objects: 34, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (21/21), done.
Writing objects: 100% (28/28), 6.87 KiB, done.
Total 28 (delta 13), reused 12 (delta 7)
To git@github.com:evildmp/afraid-to-commit.git
 * [new branch]      amend-my-name -> amend-my-name
```

Note

Be explicit!

Next time you want to push committed changes in *amend-my-name*, you won't *need* to specify the branch - you can simply do `git push`, because now *amend-my-name* exists at both ends. However, it's *still* a good idea to be explict. That way you'll be less likely to get a surprise you didn't want, when the wrong thing gets pushed.

Check your GitHub repository

- go to https://github.com/<your GitHub name>/afraid-to-commit
- check that your new *amend-my-name* branch is there
- check that your latest change to `attendees_and_learners.rst` is in it

Send me a pull request

You can make more changes locally, and continue committing them, and pushing them to GitHub. When you've made all the changes that you'd like me to accept though, it's time to send *me* a pull request.

Important: make sure that you send it from your new branch *amend-my-name* (not from your *master*) the way you did before.

And if I like your changes, I'll merge them.

Note

Keeping master 'clean'

You *could* of course have merged your new branch into your *master* branch, and sent me a pull request from that. But, once again, it's a good policy to keep your *master* branch, on GitHub too, clean of changes you make, and only to pull things into it from upstream.

In fact the same thing goes for other branches on my upstream that you want to work with. Keeping them clean isn't strictly necessary, but it's nice to know that you'll always be able to pull changes from upstream without having to tidy up merge conflicts.

Incorporate upstream changes

Once again, I may have merged other people's pull requests too. Assuming that you want to keep up-to-date with my changes, you're going to want to merge those into your GitHub fork as well as your local clone.

So:

- on GitHub, pull the upstream changes into your fork the way you did previously

Then switch back to your master branch in the usual way (`git checkout master`). Now, fetch updated information from your GitHub fork (**origin**), and merge the master:

```
git fetch
git merge origin/master
```

So now we have replicated the full cycle of work we described in the previous module.

Note

`git pull`

Note that here instead of `git fetch` followed by `git merge`, you could have run `git pull`. The `pull` operation does two things: it **fetches** updates from your GitHub fork (**origin**), and **merges** them.

However, be warned that occasionally `git pull` won't always work in the way you expect, and doing things the explicit way helps make what you are doing clearer.

`git fetch` followed by `git merge` is generally the safer option.

Switching between branches locally

Show local branches:

```
git branch
```

You can switch between local branches using `git checkout`. To switch back to the *master* branch:

```
git checkout master
```

If you have a changed tracked file - a tracked file is one that Git is managing - it will warn you that you can't switch branches without either committing, abandoning or 'stashing' the changes:

Commit

You already know how to commit changes.

Abandon

You can abandon changes in a couple of ways. The recommended one is:

```
git checkout <file>
```

This checks out the previously-committed version of the file.

The one that is not recommended is:

```
git checkout -f <branch>
```

The `-f` flag forces the branch to be checked out.

Note

Forcing operations with `-f`

Using the `-f` flag for Git operations is to be avoided. It offers plenty of scope for mishap. If Git tells you about a problem and you force your way past it, you're inviting trouble. It's almost always better to find a different way around the problem than forcing it.

`git push -f` in particular has ruined a nice day for many people.

Stash

If you're really interested, look up `git stash`, but it's beyond the scope of this tutorial.