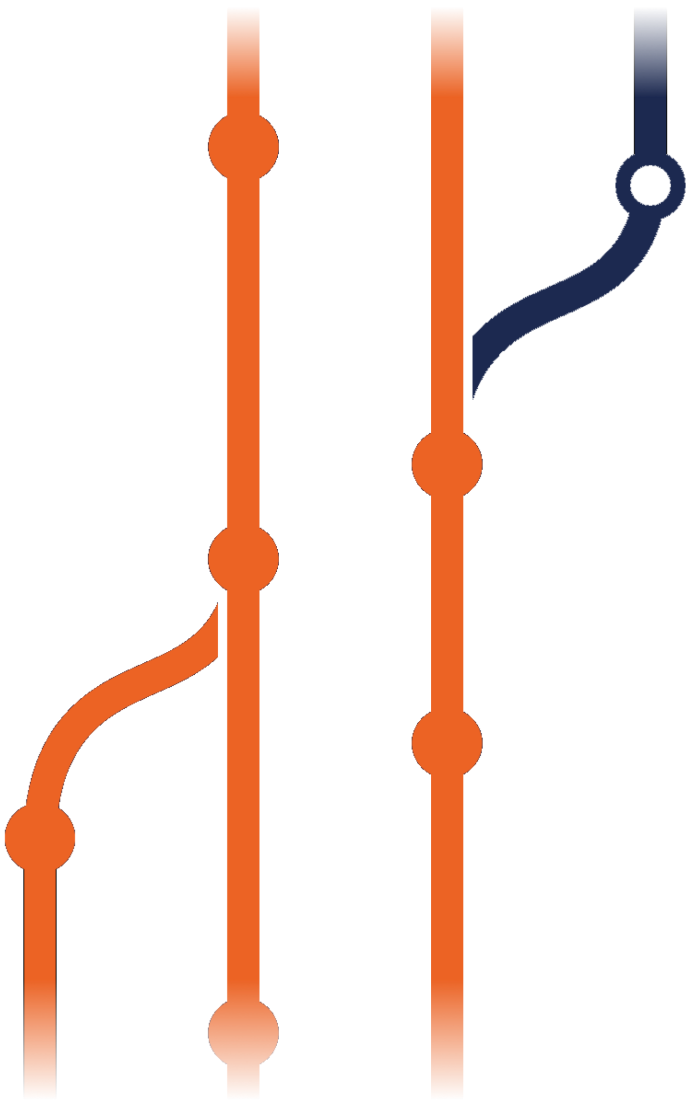
CompSoc Presents

Intro to Version Control with Git



Created and maintained by CompSoc, University of Canterbury, New Zealand (www.compsoc.org.nz).
For enquiries about academic events or this document: academic@compsoc.org.nz
For general enquiries: contact-us@compsoc.org.nz

Version 1.0 (July 2016) by Amanda Deacon, Emily Price, and Ben Moskovitz.
Version 1.1 (July 2017) by Amanda Deacon and Sam Spekreijse.
Version 1.2 (March 2019) by Patrick Ma and Jake Faulkner

A quick introduction...

Welcome to CompSoc's Intro to Version Control with Git workshop!

This workshop is aimed at anybody who wants to learn how to better manage their code. Having a clear, reliable way of managing code works wonders, whether you're crafting a solo hobby project, tackling a group assignment, or working in the big wide world.

We hope this will be particularly useful for students in ENCE 260, INFO 263, SENG 202, and SENG 302, as well as any interested first-year students.

A note of caution

These upcoming skills involve **sharing your code**. Sharing code can be a wonderful thing - there's a beautiful culture around Free and Open-Source Software (FOSS), with masses of code out there which anyone can use and even help create!

However, please be mindful when it comes to university assessments; you could land in hot water if you, for example, publicly share assignment code or course material which you weren't meant to. If in doubt, talk to teaching staff or refer to departmental regulations.

How to approach this workshop

If you're completely new to Git and version control, you may like to focus mainly on Part Zero and Part One. If you have some Git experience, feel free to dive into one of the later sections.

- Part Zero describes the theory of **version control systems**, and **Git** in particular.

- Part One covers some basic Git commands, from the ground up.

- Part Two will introduce the idea of using Git in conjunction with a hosting service - in this case, GitHub. While Part One is for individuals, Part Two is best done in pairs.

- Part Three introduces a few miscellaneous or more advanced topics, such as GitHub Flow, and using Git with IDEs.

We plan to break halfway for pizza and some socialising, however if you zoom through the handout, or take it a little slower, all good! No matter what point you're up to, when we say it's time for pizza, that includes you.

Pro tips:

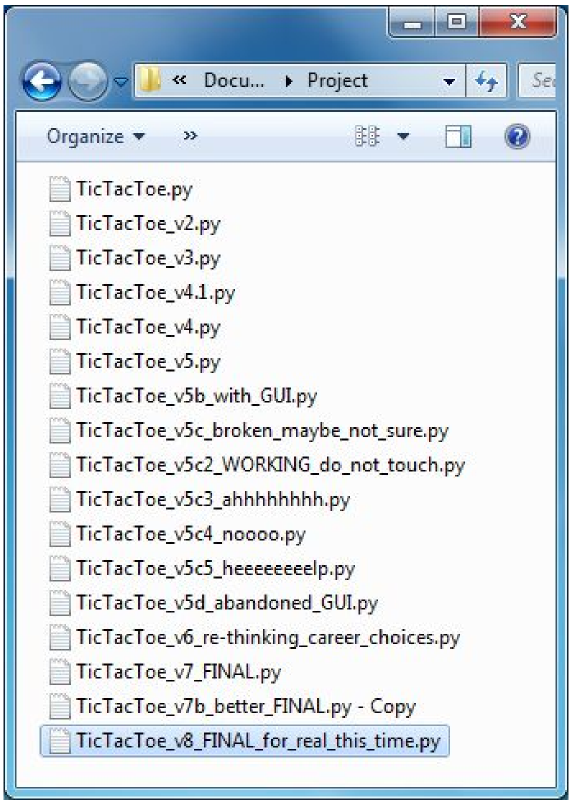
- **Take your time** - Try not to pressure yourself to whizz through everything.
- **Ask loads of questions** - We're here to help!
- **Keep learning later** - This handout will still be available to you after today.
- **Have fun!**

PART ZERO

What is this "version control" thingy anyway?

The purpose of **version control** is to record the changes made to some collection of information over time. This collection of information could be a coding project, website, set of images, or even something like a book. It's a simple premise, with awesome potential. Version control is also sometimes called **source** or **revision** control.

Take a look at this:



```
TicTacToe.py
TicTacToe_v2.py
TicTacToe_v3.py
TicTacToe_v4.1.py
TicTacToe_v4.py
TicTacToe_v5.py
TicTacToe_v5b_with_GUI.py
TicTacToe_v5c_broken_maybe_not_sure.py
TicTacToe_v5c2_WORKING_do_not_touch.py
TicTacToe_v5c3_ahhhhhhhh.py
TicTacToe_v5c4_noooo.py
TicTacToe_v5c5_heeeeeeeelp.py
TicTacToe_v5d_abandoned_GUI.py
TicTacToe_v6_re-thinking_career_choices.py
TicTacToe_v7_FINAL.py
TicTacToe_v7b_better_FINAL.py - Copy
TicTacToe_v8_FINAL_for_real_this_time.py
```

This is a method of version control! It's often where a lot of us start, and that's just fine. But, it's not a super pretty sight, nor is it clear, maintainable, or simple to share with multiple people. And it's easy to make mistakes, like accidentally saving over the wrong file. Although this kind of method can work out *okay* for smaller projects, it'll haunt your dreams if you try to do anything more complex.

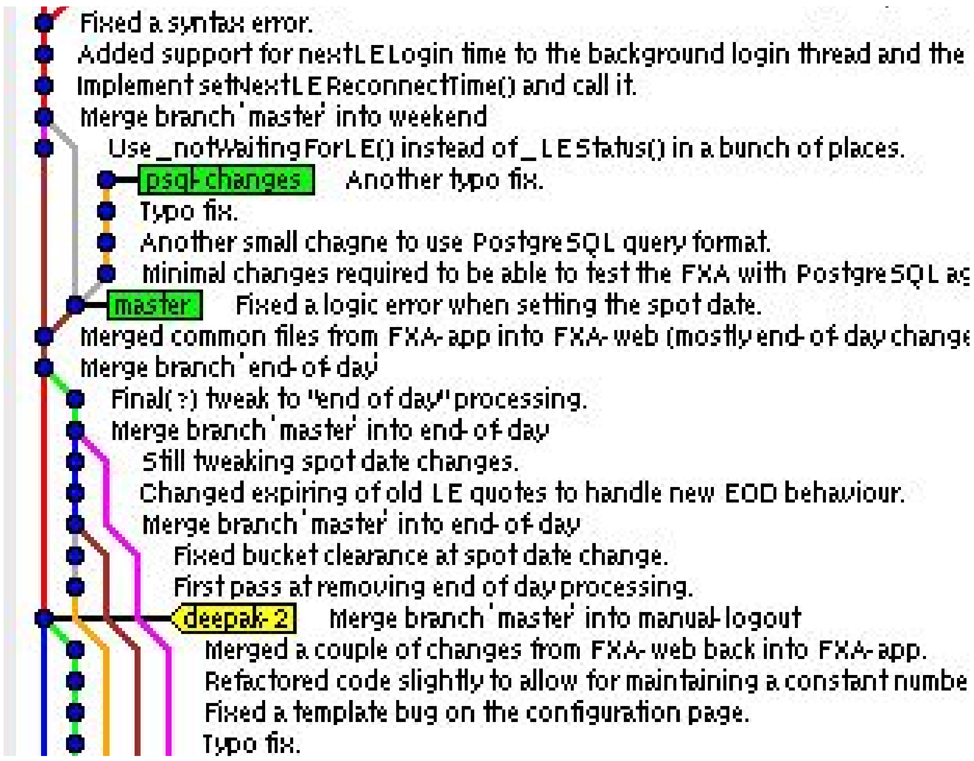
There has to be a better way ... right?

Never fear! This process can be streamlined and powerful. That's what we're here to learn about. Version control systems, such as Git, provide an elegant and reliable way to:

- **Track the changes** made to your code.
- **Re-visit previous versions** of a file or an entire project.
- **Allow multiple people** to collaborate in an organized way.

- **Create "branches" and "merges"** in your project; this is sometimes called **non-linear** development. For example, you could "branch" away from a particular line of development to explore new or experimental ideas. And if you decide to later on, this branch can be "merged" back into where you were before.

Below is an example snapshot of the sort of organizational power we can achieve. In this image of a Git project, the colourful lines in the flow diagram represent different branches of development. The comments detail the various changes that have been made over time:



```
Fixed a syntax error.
Added support for nextLELogin time to the background login thread and the
Implement setNextLEReconnectTime() and call it.
Merge branch 'master' into weekend
Use _notWaitingForLE() instead of _LEStatus() in a bunch of places.
psql-changes   Another typo fix.
Typo fix.
Another small chagne to use PostgreSQL query format.
Minimal changes required to be able to test the FXA with PostgreSQL as
master   Fixed a logic error when setting the spot date.
Merged common files from FXA-app into FXA-web (mostly end-of-day change
Merge branch 'end-of-day'
Final(?) tweak to "end of day" processing.
Merge branch 'master' into end-of-day
Still tweaking spot date changes.
Changed expiring of old LE quotes to handle new EOD behaviour.
Merge branch 'master' into end-of-day
Fixed bucket clearance at spot date change.
First pass at removing end of day processing.
deepak-2   Merge branch 'master' into manual-logout
Merged a couple of changes from FXA-web back into FXA-app.
Refactored code slightly to allow for maintaining a constant numbe
Fixed a template bug on the configuration page.
Typo fix.
```

***Figure 3**: What a time to be alive.*

So what's Git? And how does it work?

Git is a "distributed" version control system. It's free and open-source.

Extra for Experts:
Version control systems can be Local, Central, or Distributed. Read more about this here: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

The key element of Git is the **repository,** or **repo** for short. Simply put, the repository contains the entire history of a project (and some other metadata). It stores this history as a series of **snapshots**. You can access these snapshots to see exactly what a given file (or the entire project) looked like at a particular stage of its development.

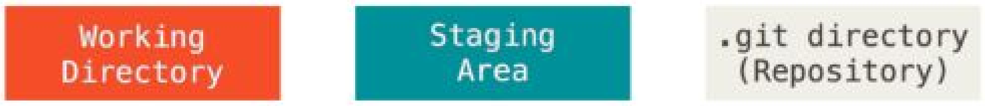
The main states of a file

There are three main states for a file in Git. These are: committed, modified, and staged. A **committed** file is one which has been stored in your local repository. A **modified** file is one which has been changed, but not committed. A **staged** file is a modified file which has been marked to be included the next time you choose to "commit" your changes to the repository.

If you're a beginner, it's totally normal for some of this terminology to sound bizarre and new. After the next chunk of reading, we'll start practicing, which will help to make sense of these concepts.

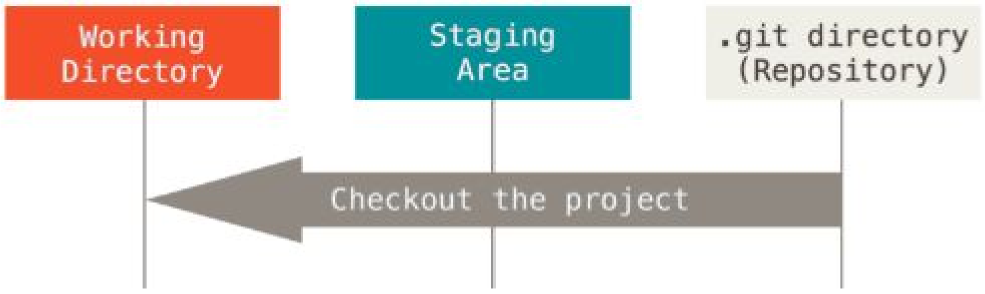
The main areas of a Git project

Above we talked about the states of **files**, which leads us neatly into describing the **areas** of a Git project. Every Git project has three main areas:

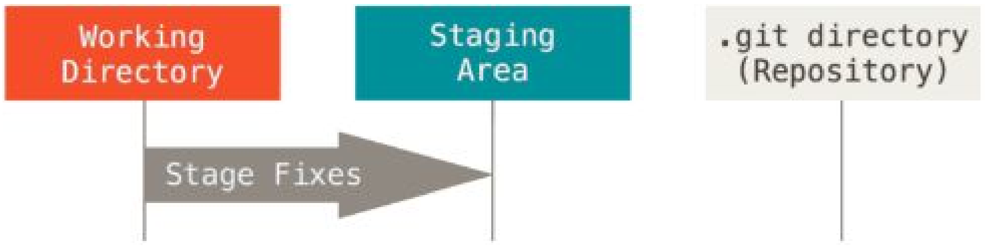


The **.git directory / repository** contains all your project metadata and other information, as described above. For this section, assume we are looking at a project in progress.

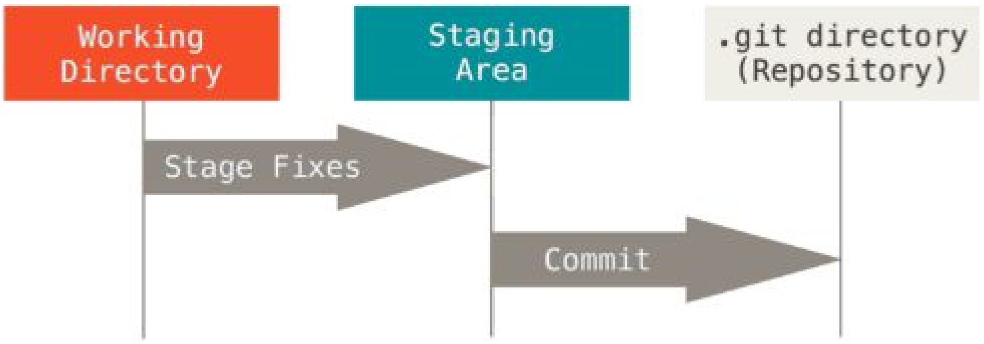
The **Working Directory** is, as the name suggests, your work area. This is a "checkout" of one version of the project. These files get pulled out of the repository and placed on disk for you to modify.



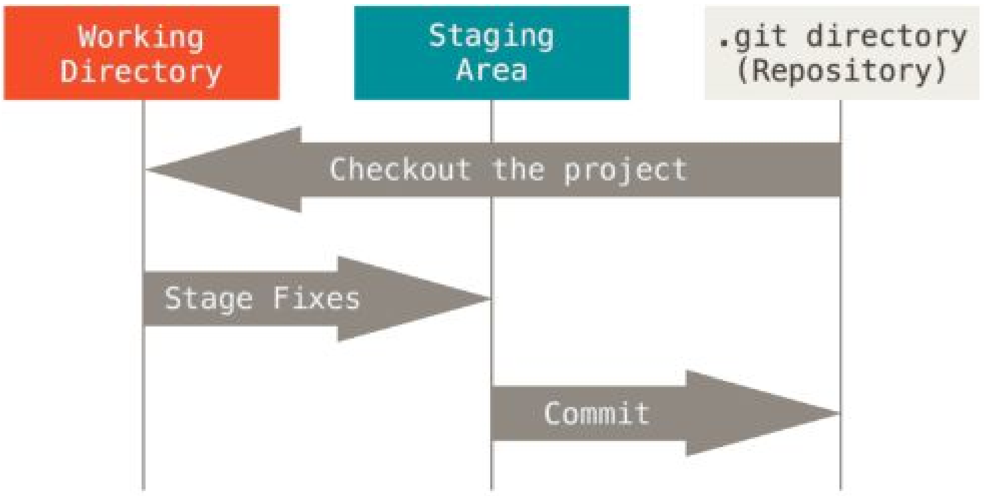
The **Staging Area** is a sort of middle ground. As you modify files in the Working Directory, you can place (or stage) these files into the Staging Area.



Once you're ready, you can commit these changes, which means the files are taken from the staging area and and stored permanently as a snapshot in your repository.



And putting everything together:



So what's the relationship between file states and project areas?

A particular version of a file is in a **committed** state if it exists in the repository. A file which is chilling out in the staging area is considered **staged**. A file which has been checked out of the repository and modified (but not staged) is considered **modified**.

Summary

Woohoo, you made it through Part Zero! Remember, this information is here to give you a general overview of what's going on, and you don't have to take it all in at once. You may find it helpful to refer back to this section once in awhile. Now, let's put these ideas into action.

PART ONE

Here, we'll learn some fundamental Git commands.

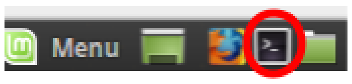
First steps

Installing Git

All of the Erskine lab computers will already have Git installed, so hooray! However, if you're working from home and need to install or update Git, Windows / Mac / Linux users can head here for some tips: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Terminal setup

Open up a terminal window. You can do this by pressing `CTRL+ALT+T` on the lab computers or pressing the little >_ button on the taskbar:



Navigate into a folder such as `Documents`. Within here, create a new folder called `MyProject`, and then navigate into it. The commands might look like this:

```
cd Documents/
mkdir MyProject
```

Terminal Tips:
New to terminal? Here are some helpful commands:

- `mkdir MyProject` will create a new folder called `MyProject`.
  - `mkdir` = make directory
  - "Directory" is a fancy-sounding word for "folder".
- `cd MyProject` will navigate you into the `MyProject` folder.
  - `cd` = change directory
- `ls` (that's ell, ess) lists the contents of the current directory.
- `cat <filename>` displays the contents of a file.
- Pressing the up/down arrow keys will show your command history.

You can also use CompSoc's handy-dandy terminal cheatsheet if you want to.

First-time Git setup

When you start using Git for the first time, you should set your username and email address. Enter the following commands into the terminal (using your own details):

```
git config --global user.name "Jellybeans McLovin"
git config --global user.email jellybeans@example.com
```

The `--global` option means that Git will always use this information for anything you do on this computer. You only need to do this process **once** on a given computer.

To display your current settings, enter:

```
git config --list
```

Use this now to check that your username and email have been correctly recorded.

Getting help

You can get help on particular Git commands from within the terminal. Any of the following will display the relevant manual pages for a command:

```
git help <verb>
git <verb> --help
man git <verb>
```

For example, `git help config` will bring up the manual pages for the config command.

Initializing a repository

There are two different ways to get a Git project going. We can:

- **Option One:** "Clone" an existing repository.
- **Option Two:** Initialize our own repository.

Here we'll go with Option Two, and start our own repository (later on in the handout, we'll learn about Option One). In your terminal window, you should be located in the `MyProject` directory (if not, please re-visit the Terminal Setup section). Enter the following command:

```
git init
```

This will create a subdirectory (a folder within your current folder) named `.git`. This is your shiny new **Git repository**.

Extra for Experts:
Head here for an interesting discussion of the exact contents of a Git repository: https://www.sbf5.com/~cduan/technical/git/git-1.shtml

Tracking new files

Let's create a couple of files, and get some practice with basic Git commands.

Always make a readme

It's good practice for every Git project to contain a **readme** file, which contains a description of the project. Create a new file containing the following:

```
My first Git project.
Author: <your name>
Date started: <date>
```

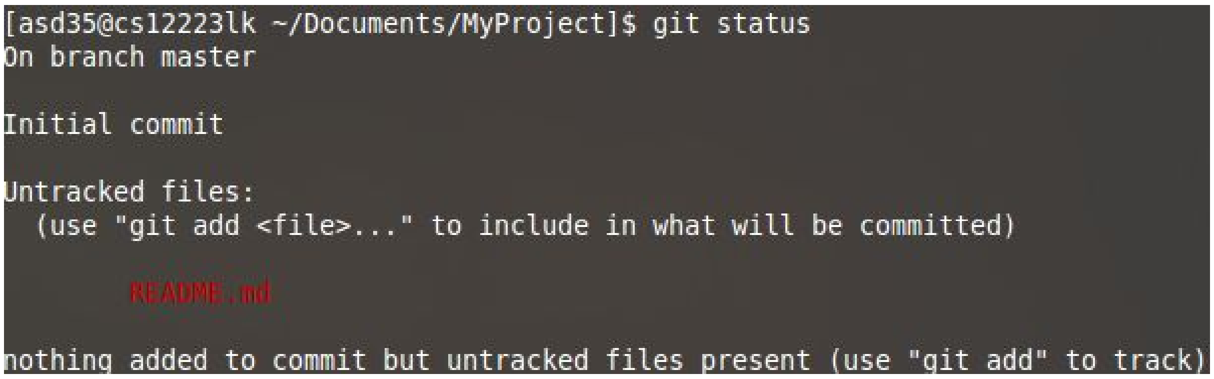
Save this file as `README.md` within the `MyProject` directory.

> **Markdown Files**
> Markdown files, which have the extension `.md`, are an easy way of formatting text on the web without faffing about with HTML . More information here.

Now let's check the status of our project with the following command:

```
git status
```

You'll see that the `README.md` file is currently "untracked". Hmmm:

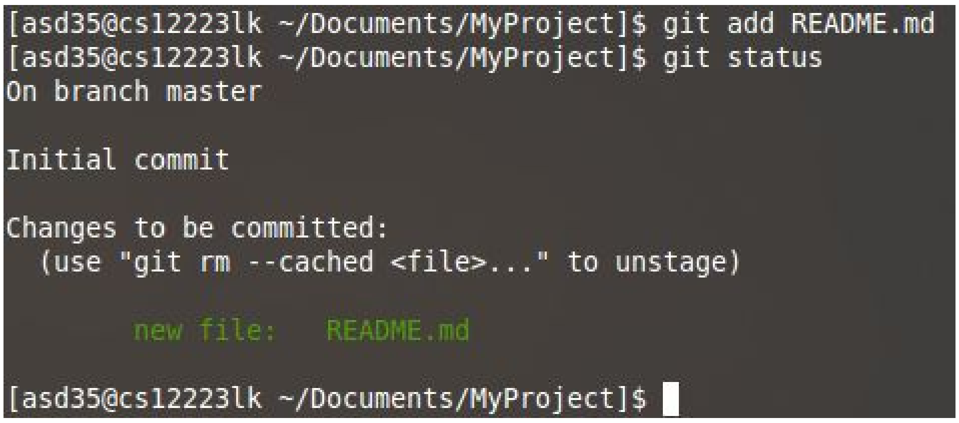


```
[asd35@cs12223lk ~/Documents/MyProject]$ git status
On branch master

Initial commit

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

        README.md

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

It turns out that Git won't automatically keep track of this file for us. We need to explicitly tell it to do so with the following:

```
git add README.md
```

Now, running `git status` again shows us that `README.md` is now tracked, and has been placed in the staging area.



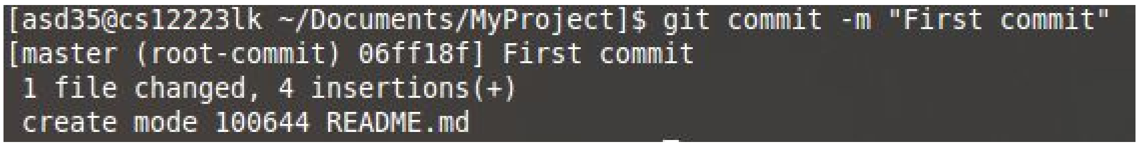
```
[asd35@cs12223lk ~/Documents/MyProject]$ git add README.md
[asd35@cs12223lk ~/Documents/MyProject]$ git status
On branch master

Initial commit

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

        new file:   README.md

[asd35@cs12223lk ~/Documents/MyProject]$
```

First commit

To commit the currently staged `README.md` to the repository, enter:

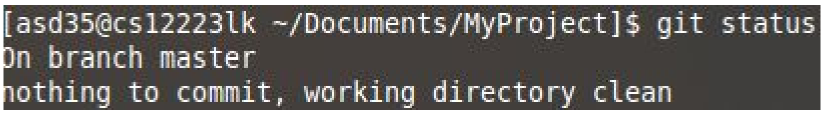
```
git commit -m “First commit”
```

The `-m` allows us to add an accompanying message in-line, instead of via a text editor. If you left it out, you'd be stuck in vim, and no one wants that. A snapshot of this current state of the project has now been stored permanently in the repository. Within the terminal, you'll see a summary of the changes you've made in this commit:



```
[asd35@cs12223lk ~/Documents/MyProject]$ git commit -m "First commit"
[master (root-commit) 06ff18f] First commit
 1 file changed, 4 insertions(+)
 create mode 100644 README.md
```

Now, a quick status check will show that:



```
[asd35@cs12223lk ~/Documents/MyProject]$ git status
On branch master
nothing to commit, working directory clean
```

Modifying files

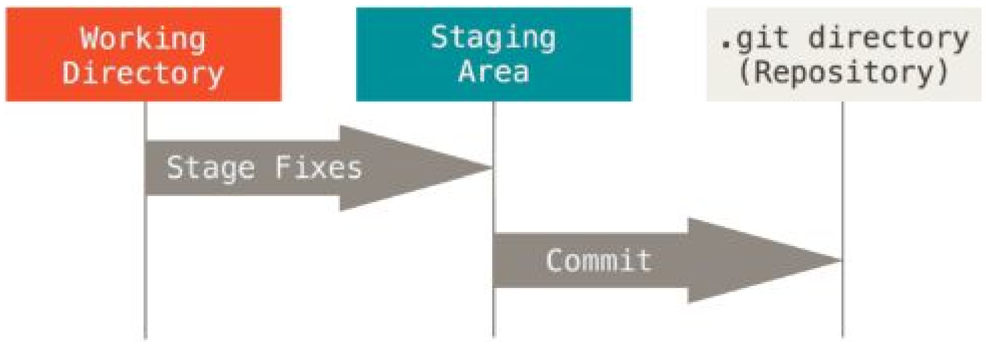
Now, create a new file with the following contents:

```
How I wish that I could be
a happy humming bumblebee.
```

Save your masterpiece within the `MyProject` directory as `poem.txt`. Next, make two commands in sequence:

1. Tell Git to **track** this file, using the `add` command.
2. Do a **commit** with the `commit` command and the message “Add a poem”.

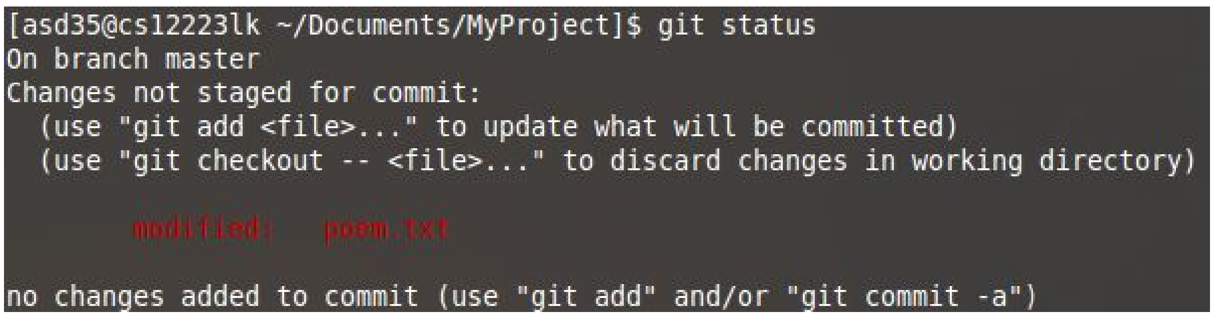
Review these commands in the previous sections if needed. As a visual reminder of what we're doing, here is a diagram from Part Zero:



Now, make a change to the poem. For example, alter the second line to read:

```
How I wish that I could be
a bumbling mumbling manatee.
```

Save these changes (under the same name of `poem.txt`). On a `git status` check, we'll see that `poem.txt` is marked as **modified**. Note that it hasn't been added to the staging area yet:



```
[asd35@cs12223lk ~/Documents/MyProject]$ 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:   poem.txt

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

Now, **stage** the file using the `add` command, and then do a commit with this command:

```
git commit -m "Poem fix: Change bumblebee to manatee."
```

<u>**Note 1:**</u> We have used the add command in two different contexts. Firstly, we used it on newly created files; here the command causes the file to become tracked as well as staged. Secondly, we use it a file which was already tracked but had been modified; here, the command will just stage the file.

<u>**Note 2**</u>: If you ever stage a file with `git add`, but then modify the file, you'll need to run `git add` again to stage the latest version.

Viewing changes

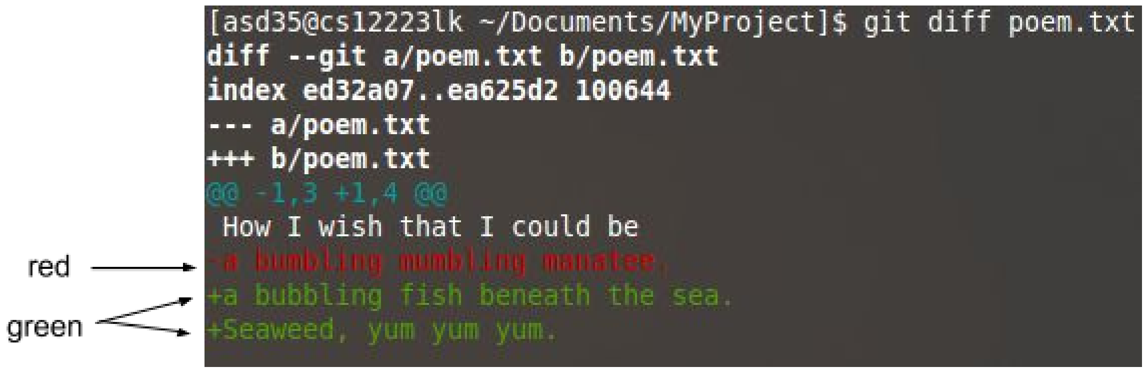
Let's modify the poem again. Modify the second line, and third one, so that the poem reads:

```
How I wish that I could be
a bubbling fish beneath the sea.
Seaweed, yum yum yum.
```

You can use the following command to compare these modifications to your last commit:

```
git diff poem.txt
```

In the screenshot below, the red lines (preceded by minus signs) indicate lines which have been deleted, and the green lines (preceded by a plus sign) indicate lines which have been added:



```
[asd35@cs12223lk ~/Documents/MyProject]$ git diff poem.txt
diff --git a/poem.txt b/poem.txt
index ed32a07..ea625d2 100644
--- a/poem.txt
+++ b/poem.txt
@@ -1,3 +1,4 @@
 How I wish that I could be
-a bumbling mumbling manatee.
+a bubbling fish beneath the sea.
+Seaweed, yum yum yum.
```

Now, stage this new version of the poem. Try running the git `diff poem.txt` command again. Notice that nothing happens this time - weird! This is because, if you want to compare a **staged** file to your last commit, the command needs to be altered slightly:

```
git diff --staged <filename>
```

These are both subtly different. `git diff <filename>` lets you see what you have modified but not staged. On the other hand, `git diff --staged <filename>` compares a staged file to its last commit.

Discarding changes from a modified file

Once again, let's alter the poem:

```
How I wish that I could be
a world-destroying centipede.
Doom doom destruction raaaaaah!!!
```

Save this under the same name of `poem.txt` . But, what if we decide that we really don't like these changes, and just want to get rid of them entirely? After all, you probably don't want to be a world-destroying centipede. If you modify a file (but haven't staged or committed it yet), and then decide that you want to discard those changes, you can enter:

```
git checkout -- poem.txt
```

Run this command now. Then, open up `poem.txt` . You'll see that the centipede changes are no longer there, and we are back to our bubbling fish version. This command has updated the file to match up with the bubbling fish version which we'd previously committed (there's a bit more technical detail, but we'll stick with this explanation for now).

When used in this context, the `checkout` command can be really dangerous, so watch out! Be absolutely sure that you want to do it, because this particular process isn't reversible. The centipede version of the poem is now gone for good.

Now, `commit` the bubbling fish version of the poem, using the commit message "Poem fix: Change manatee to fish".

Unstaging a staged file

If you have staged a file, but would like to unstage it, you can enter:

```
git reset HEAD <filename>
```

Extra for Experts - Other ways of undoing stuff:
With Git, there are many, many ways of doing (and undoing) things. We can't cover everything here, but if you'd like more detail on undoing actions, check out this page of the Git documentation here.

Skipping the staging area

Sometimes, instead of modifying a file, adding it to the staging area, and then doing a commit, you may feel like skipping that middle staging step entirely. Git allows you to take this shortcut with the following command:

```
git commit -am "Message goes here"
```

Note that `-am` has the same effect as `-a -m`.

The `-a` stands for "all". This single command will take **every changed file**, whether or not it's already in the staging area, and commit this snapshot to the repository. This shortcut can be really useful, but be careful that you aren't accidentally committing unwanted changes. Feel free to create a few new files and practice this now.

File removal

Removing files from Git, but leaving them in your working directory

Sometimes, you may want to keep a file on your hard drive, but remove it from Git. Use the `cached` option to do this:

```
git rm --cached <filename>
```

Removing files from both Git and your working directory

Other times, you may want to remove a file from both Git and your local drive. To remove a file from Git, you first tell Git to stop tracking it, then do a commit. So, the following two commands in sequence are needed:

```
git rm <filename>
git commit -m "Message"
```

Nuking everything from orbit - AKA, Oh God I messed everything up and I need to go back to the last commit

Sometimes, you really stuff something up when you're coding. That's okay! Not only does it happen to all of us, it's also pretty easy to get back to the way things were. Just use these two commands:

```
git stash
git stash drop
```

The first command puts all of your uncommitted work away for safe-keeping, and the second one unceremoniously deletes it.

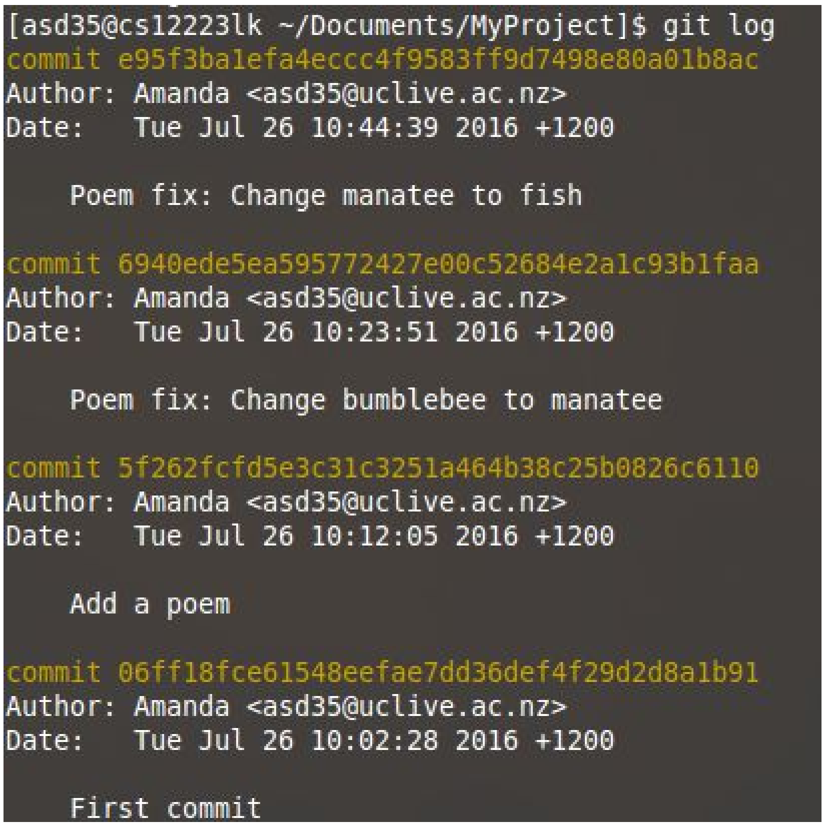
Commit history

Looking at your commit history

You can view your commit history / log. Enter the following command:

```
git log
```

This allows you to see each commit made on the project, accompanied other details of the commit. Here, the commits are listed beginning with the most recent:



```
[asd35@cs12223lk ~/Documents/MyProject]$ git log
commit e95f3ba1efa4eccc4f9583ff9d7498e80a01b8ac
Author: Amanda <asd35@uclive.ac.nz>
Date:   Tue Jul 26 10:44:39 2016 +1200

    Poem fix: Change manatee to fish

commit 6940ede5ea595772427e00c52684e2a1c93b1faa
Author: Amanda <asd35@uclive.ac.nz>
Date:   Tue Jul 26 10:23:51 2016 +1200

    Poem fix: Change bumblebee to manatee

commit 5f262fcfd5e3c31c3251a464b38c25b0826c6110
Author: Amanda <asd35@uclive.ac.nz>
Date:   Tue Jul 26 10:12:05 2016 +1200

    Add a poem

commit 06ff18fce61548eefae7dd36def4f29d2d8a1b91
Author: Amanda <asd35@uclive.ac.nz>
Date:   Tue Jul 26 10:02:28 2016 +1200

    First commit
```

There are loads of options you can add to the `git log` command, depending on what you're looking for. A particularly awesome one is the `-p` option. This shows the differences introduced in each commit. Try this option out now.

If you'd like to see more options to go with `git log`, remember that you can browse the manual pages for any particular command. Re-visit the Getting help section if you aren't sure how to do this.

Practice time!

Time to review these new skills. Take five or ten minutes to create a couple of new files and play around. And don't worry about breaking anything! Can you:

- Track new files?
- Modify, stage, and unstage files?
- Check the status of the project?
- View differences?
- Commit your changes?

If you're working with our accompanying worksheet, now is a good time to annotate it with the information needed to perform the above actions.

PART TWO

Remote repositories

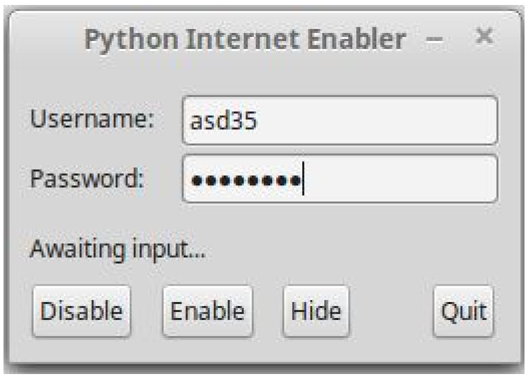
Up until now, we've been doing everything locally (i.e. on our individual computers). But, if we want to collaborate with other people, it makes sense to be able to store a project repository in multiple places. A **remote repository** is a version of your project which is hosted on the Internet (or somewhere on your network).

For this section, we'll be working with a Git host called **GitHub**. There are many different things you can do with Git and GitHub, and we can't explore them all today. We'll just be looking at a few aspects today, as a small taster of what you can do. We'll continue working with the terminal (rather than the GitHub GUI) because it's a much more transferable skill.

Try to follow these instructions closely when you're first learning. Once you get more confident, feel free to experiment!

Python Internet Enabler

On the Erskine lab computers, you'll need turn on Python Internet Enabler. You can search for this in the Start Menu. Enter your university username and password, and click "Enable":



Username: asd35
Password: ●●●●●●●●●
Awaiting input...

Create a GitHub account

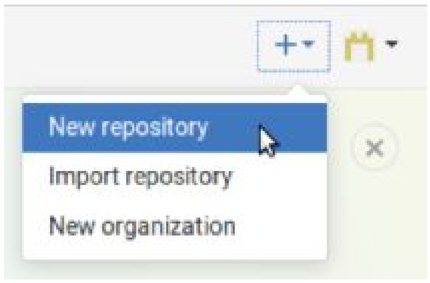
Head to https://github.com and create an account. The signup process has three stages:

- **Step 1:** Choose a username and password. Enter an email address (personal or university email - either is fine).
- **Step 2:** In the "Choose your plan" section, select the free option.
- **Step 3:** Skip or answer the survey questions.

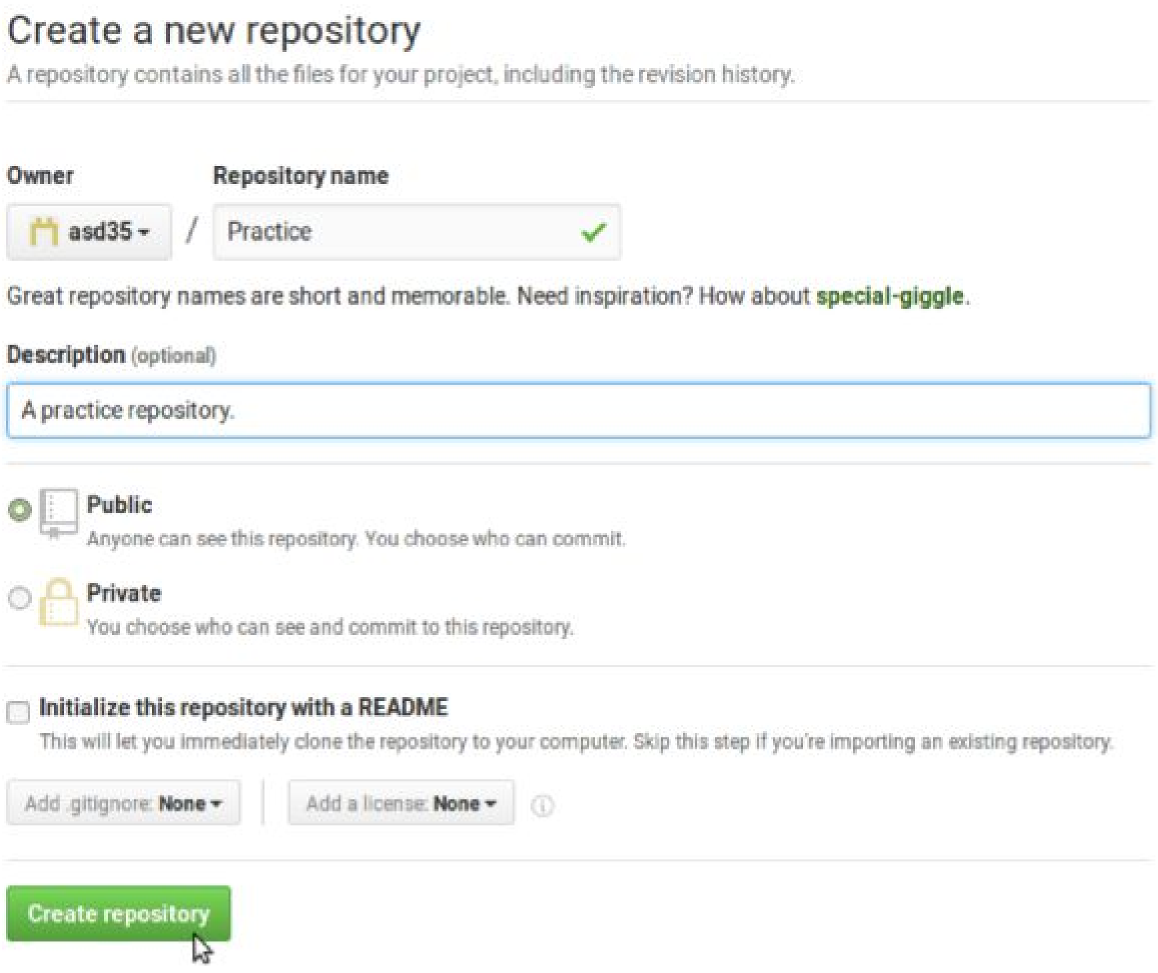
Note: "Git" is not the same thing as "GitHub". Git is a piece of version control software, whereas GitHub is a place to store your Git repositories online. Other Git hosts include GitLab - which you might run into in some second- or third-year computer science courses - and BitBucket.

Create a new repository

Use the toolbar in the top right of the GitHub page to create a new repository:



Enter a name and description for your project, and then click **Create repository**.


Create a new repository

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

Owner: asd35 / Repository name: Practice

Great repository names are short and memorable. Need inspiration? How about special-giggle.

Description (optional): A practice repository.

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

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

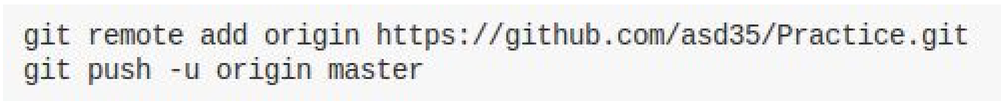
Initialize this repository with a README

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

This has created a new empty repository, hosted by GitHub. You should end up on a **Quick setup** page, which offers you a few options for what to do next. We'll need the two lines of code which look something like this (your exact URL will differ):



```
git remote add origin https://github.com/asd35/Practice.git
git push -u origin master
```

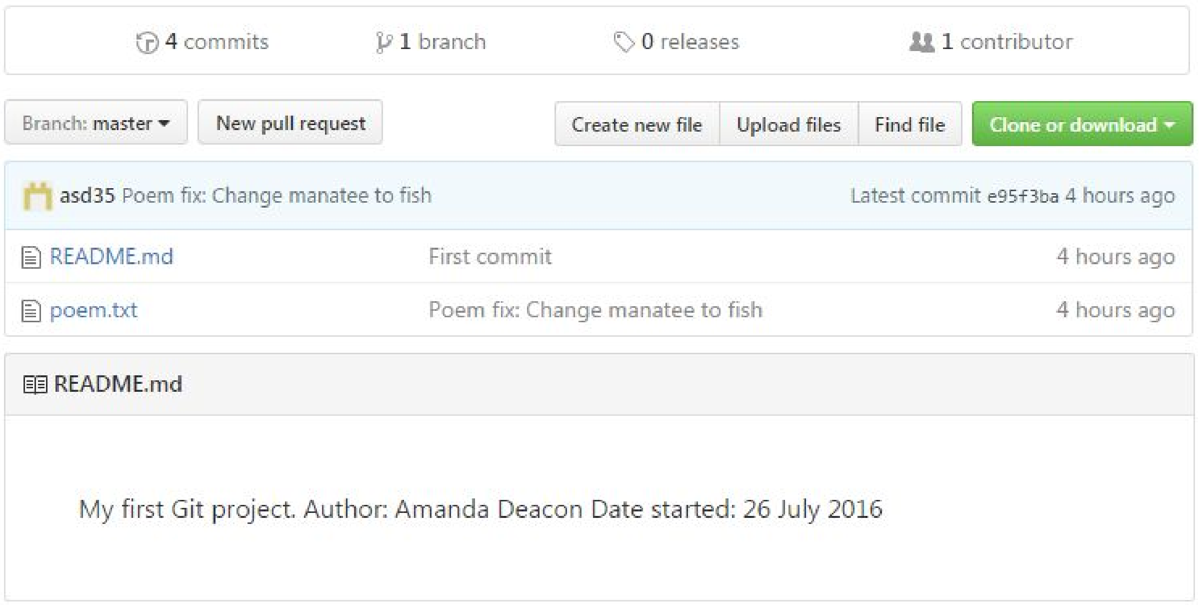
Pushing an existing repository from the command line

So far, we've been working in the `MyProject` directory, with all our project information stored only locally. We're going to **push** this local information up into the empty repository we just created on GitHub.

In your terminal window, make sure you're located within the `MyProject` directory. Then, enter the two commands from the screenshot above, making sure to customize them with your own details:

```
git remote add origin https://github.com/username/project.git
git push -u origin master
```

Enter your GitHub username and password when prompted by the terminal. Now back to your web browser. Refresh the current GitHub page, and you'll hopefully see that your `MyProject` files have appeared, like this:



asd35 Poem fix: Change manatee to fish — Latest commit e95f3ba 4 hours ago

README.md — First commit — 4 hours ago

poem.txt — Poem fix: Change manatee to fish — 4 hours ago

README.md

My first Git project. Author: Amanda Deacon Date started: 26 July 2016

Okay, so what just happened?!

GitHub is now hosting this project and its entire history. This is awesome! Now, your project is stored in more than one place, so it's less likely that you'll have trouble with unexpected losses of data. And, with your project hosted on the Internet, it's super easy to collaborate with other people.

Friendship is magic

Adding collaborators

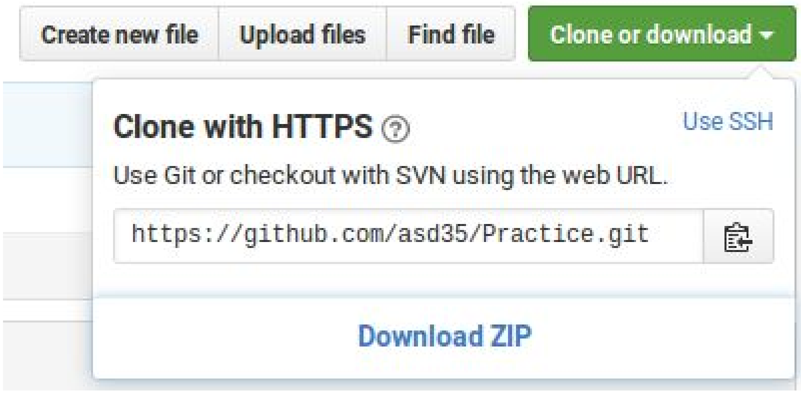
Work in pairs for this part. Ideally, read and follow these instructions together. For simplicity, label yourselves **Person 1** and **Person 2**.

Person 1: Let's say Person 1 has a repository hosted on GitHub. Use whichever name you gave your repository in the previous steps - these instructions will refer to this as the "Friendship" repository. Let's also say you want to allow Person 2 to make changes to this repository.

Person 1: Within the GitHub page for the Friendship repository, go to Settings >> Collaborators. Add Person 2 (using Person 2's GitHub username) as a collaborator.

Person 2: You should soon receive an email, inviting you to join Person 1's repository as a collaborator. Open up this email and accept the invitation.

Person 2: You'll now find yourself on the GitHub page for the Friendship repository. Click on the **Clone or download** button, and copy this URL - you'll need this for the next step.



```
https://github.com/asd35/Practice.git
```

Person 2: Use the terminal to navigate to your **Documents** folder. Now, let's **clone** Person 1's project with the following command:

```
git clone URL
```

making sure to replace `URL` with the URL you copied in the previous step. You may be prompted to enter your (GitHub) username and password.

This `clone` command will automatically create a folder using the name of the repository, "Friendship". All of the needed files and data (including the entire history of the project) will now be in there. Within the Friendship folder, use the `ls` command to see that this is indeed the case. Magical!

Person 2: Now, make some sort of change to the project. For example, create a new file, or alter an existing file. Stage and then commit these changes. So far, this has only altered the project on your local drive; we need to take another step to **push** these local commits to any associated remote repositories (in this case, the repository on GitHub). Enter the following:

```
git push
```

This will update the GitHub repository with your local changes. Here we've taken quite a simple approach. This process can get a bit messy if other collaborators have made changes to the GitHub repository in the meantime. We'll talk about this more in Part Three.

Person 1: Within GitHub, have a look at the Friendship repository - you may need to refresh the page. If all goes well, you'll see that Person 2's changes will have appeared. Hooray!

Phew, that's it for Part Two!

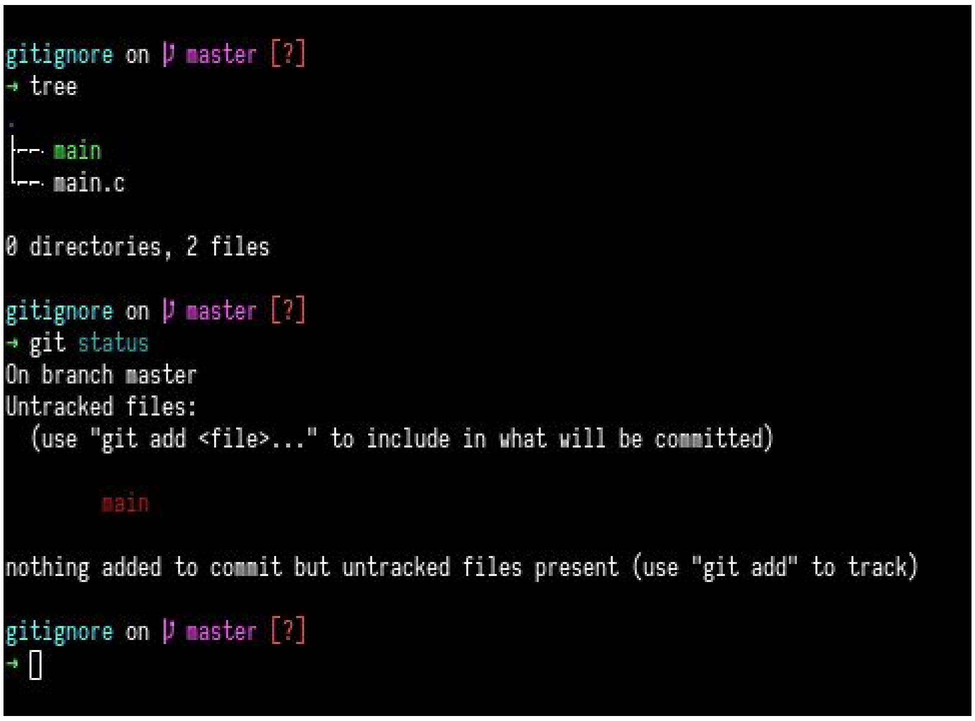
Congrats! If you're confused, now is a good time to ask questions. Otherwise, the party continues in Part Three.

PART THREE

This section can be considered either “extras for experts”, or “random interesting topics that didn’t fit elsewhere”.

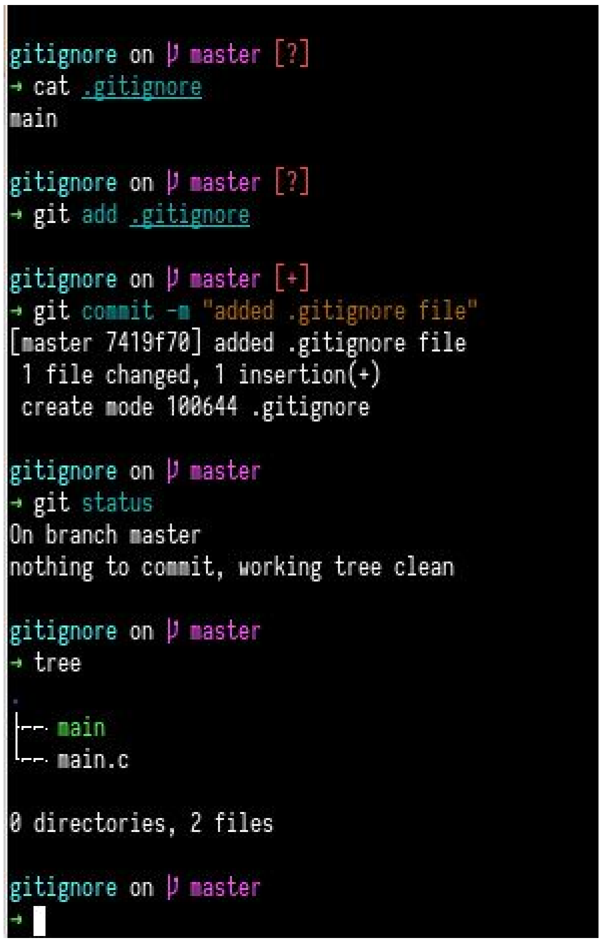
Gitignore

Just like a desk in real life your local repository can collect cruft overtime. IDE project files, object files, executables and a host of other generated junk is created in the process of working in a repository, and whilst Git will nag your about staging these new files you don’t always want everyone else having a copy of your IDEs project settings. What we’d like to be able to do is tell Git that IDE project files should not be tracked, but it should still track our important source code. It’s for issues like this that Git allows for a *.gitignore* file at the root of the repository specifying which files Git should pretend don’t exist. To see how this file works and how to write one let’s consider a really simple repository.



```
gitignore on  master [?]
→ tree
.
├── main
└── main.c

0 directories, 2 files

gitignore on  master [?]
→ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        main

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

gitignore on  master [?]
→ 
```

This is a simple C project (don’t panic if you’ve not encountered C before you don’t need any understanding of how it works). The code has been written and compiled into an executable file *main*. Because that executable is specific to my machine I’d like to tell Git that it should ignore *main* and stop nagging me about it in the status output.



```
gitignore on ᚠ master [?]
→ cat .gitignore
main

gitignore on ᚠ master [?]
→ git add .gitignore

gitignore on ᚠ master [+]
→ git commit -m "added .gitignore file"
[master 7419f70] added .gitignore file
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore

gitignore on ᚠ master
→ git status
On branch master
nothing to commit, working tree clean

gitignore on ᚠ master
→ tree
.
├── main
└── main.c

0 directories, 2 files

gitignore on ᚠ master
→ 
```

Here we've added a single line "main" to a file *.gitignore* made at the root of my project, and after committing those changes to the repository git no longer cares about the executable in the project. The syntax of a *.gitignore* file is pretty simple, each line contains a filename/path (where paths are given relative to the repository root). Filenames can contain wildcards so if you wanted to ignore all logging files you might add a line like "*.log" and Git would ignore all files ending with the ".log" extension.

GitHub Flow

So, you and your friends are hosting your amazing ~~bumblebee centipede manatee~~ fish poetry in the cloud, and you're all getting really into it. Your poems are awesome, and everyone is flocking to your GitHub profile to read them. The question is, how do you handle multiple people editing your poetry? Sure, Git and GitHub *allow* lots of people to work on the same thing, but what's the *best* way to do it?

Enter: GitHub Flow. GitHub flow is a way of using Git that minimises the amount that you step on your collaborator's toes, and makes it easier to make good software.

GitHub documents their own stuff better than we do
GitHub has a really cool little blurb about the way GitHub flow works here. It even has nifty interactive diagrams.

Branch

The basic idea is that when you want to make a change to your project, you create a new **branch**. Branching is kinda like copying and pasting your code somewhere else so you won't stuff up anyone else's work, but Git makes it really easy to switch what branches. To see what branches you have on your machine, use the command:

```
git branch
```

And to switch to a different branch, use:

```
git branch BRANCH_NAME
```

Finally, to create a new branch and switch to it, use:

```
git checkout -b BRANCH_NAME
```

Where BRANCH_NAME is whatever you want to call your branch - it's usually best to name the branch a summary of the changes you want to make. For example, you might call your branch `gui-usability-improvements` or `add-manatees`.

Commit

Once you've made your branch, work on your project as you normally would - add commits to the project, trying to not have too many lines of code in each commit. You begin moving your poetry to the a manatee-based subject matter, which you feel will benefit the project.

Pull Request and Discuss

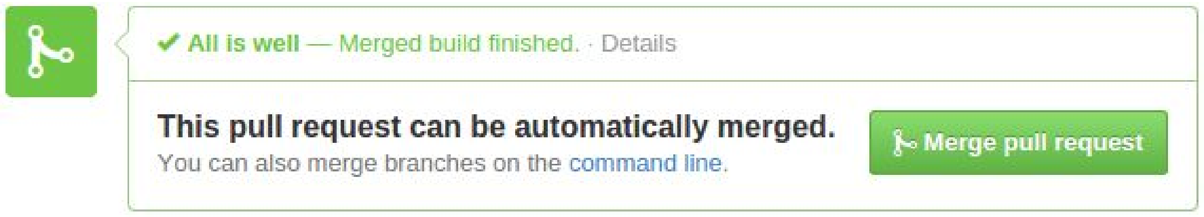
Once you feel that your changes are almost ready, you submit a **Pull Request** on GitHub. A Pull Request, or PR for short, is you saying to the rest of the people working on your project that you feel like your branch is ready to be integrated into the **master** (main) branch. Once you've submitted your pull request, one or more people working on your project will then take a look at the changes you've made, and approve or deny them.

Pull Request Nomenclature
Pull Requests are called different things in different places - on GitLab they're called **Merge Requests**, at Google they're called **Change Lists**, and many companies call them different things internally.

The idea here is to get as many pairs of eyes as possible on the code, and to try to make sure no one is introducing any *really* terrible bugs into the project.

Merge

Everything is awesome! Your friends have checked your changes to your poetry, and the gradual move away from fish and back to manatees, which you all agree are far superior, has been approved. All you need to do to integrate your manatee-related changes into master is to click the 'Merge Pull Request' button!



✔ All is well — Merged build finished. · Details

This pull request can be automatically merged.
You can also merge branches on the command line.

Unless....

Merge Conflicts

Git is great at merging code, so long as no two people have touched the same line or lines of code. This is where Git can get a little hairy, and you'll probably hear a lot of whinging on the Internet about resolving merge conflicts. Don't worry! It's really not that hard.

First things first: Make sure your version of master is the most recent one, using the command:

```
git pull origin master
```

Then, merge master into your branch. This may seem counter-intuitive, but once master is merged into your branch, you'll be able to merge your branch into master without conflicts. Now merge master into your branch using the command:

```
git merge master
```

Git will now complain about which files have conflicts. In your favourite text editor, open up the file/s Git complained about. Somewhere in them, you'll see something like this:

```
<<<<<< HEAD
(The changes you made)
==================================
(The changes that're in master)
>>>>>>> refs/heads/master
```

This is your **conflict** - the thing that Git doesn't quite know. You can either delete one of the things there, or mix them both into one - the idea here is to use your human intelligence to mix the branches in a meaningful way.

Using IDEs with Git

If, like some of us, you prefer working in IDEs (rather than editing your code in Vim...), you might be pleasantly surprised to find that certain development environments have built-in support for Git and other version control systems.

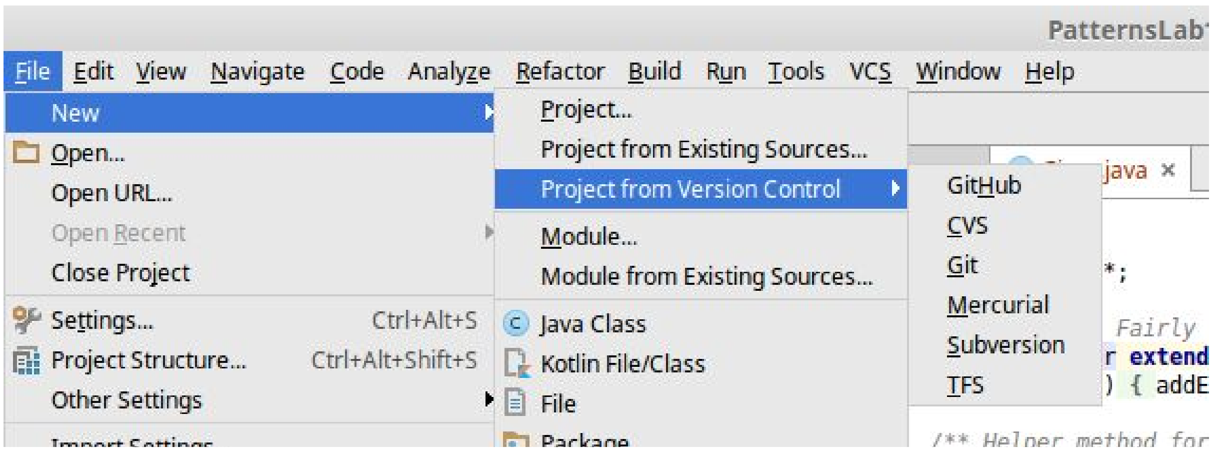
Another writer of this handout checking in: Vim is and always will be the best text editor. Don't let their lies corrupt your opinions.

If you're studying software engineering, or like writing Java, you may have heard of IntelliJ. It's a pretty slick IDE that some of us really love, due to how much it holds your hand through the Git process. If your interactions with Git are pretty much limited to pushing and pulling, or you generally prefer a GUI because the command line is hard, something like this might be for you!

JetBrains' Guide to Using Git Integration
JetBrains, the company that develops IntelliJ, has very detailed references on how to use all sorts of version control with their product. Here's a link to the Git guide.

Getting Your Remote Repo

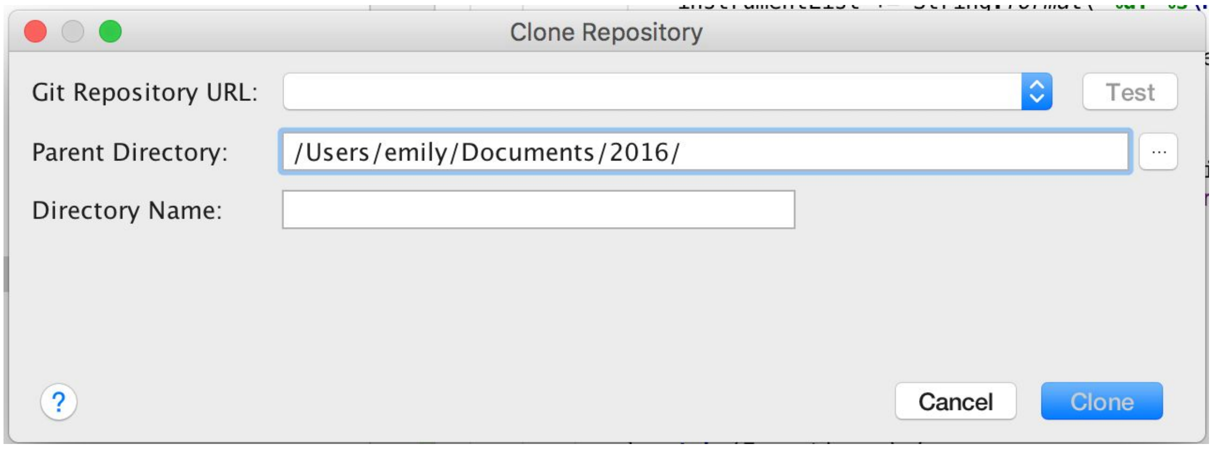
To start off with, we're going to use IntelliJ to load up the GitHub project you've been working on. To open a Git project in IntelliJ, we want to use the following menus:


PatternsLab
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
New
Open...
Open URL...
Open Recent
Close Project
Settings... Ctrl+Alt+S
Project Structure... Ctrl+Alt+Shift+S
Other Settings
Import Settings
Project...
Project from Existing Sources...
Project from Version Control
Module...
Module from Existing Sources...
Java Class
Kotlin File/Class
File
Package
GitHub
CVS
Git
Mercurial
Subversion
TFS

Although you might be tempted to click "GitHub", today we're going to be using the just straight Git option. Using this option teaches you a way that you can combine Git and IntelliJ no matter what Git host you use. This is especially helpful for those of you taking or intending to take SENG202 or ENCE260. In those classes, the university will host your projects on a

GitLab (rather than GitHub) server, so learning the GitHub-specific way to do things isn't as effective as this. We all want to learn a broad range of skills, right?

A dialog box will pop up that should look something like this:



Git Repository URL:

Parent Directory: /Users/emily/Documents/2016/

Directory Name:

Essentially what this dialog is doing:

- The first input is asking for the URL of a Git project you own. If you need a refresher on finding your URL, look back to the "Friendship is Magic" section.
- The second input is asking where you want to store your local copy, on your computer.
- The third input is asking what you want to name the folder in which you will be storing your copy of the repo.

Once you've got all this information sorted, hit "clone". IntelliJ will do some work behind the scenes - depending on the size of your repository, this could take a few moments. Once it's done, all the files you've created and edited during this lab should show up.

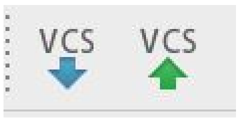
Making Changes

Say we use IntelliJ to make a change to one of our files. With Git integrated into our IDE, the IDE is kind enough to use a nice visual display of the status of all your files!

- Files that are up to date with the repo are shown in black
- Files that have been edited (and so are different to the repo version) are shown in blue
- A file that you've created in your local directory, that hasn't been pushed to Git is green
- And a file that is shown in **red** has scary merge conflicts…

Now that you've made changes, there are a bunch of different ways to go about committing and pushing them. Here's what we reckon is a pretty quick, easy way.

Firstly, we want to look for these guys:



They're probably hanging out around the top right corner of your screen (but talk to us if you can't find them anywhere). For making changes, we want to use the one on the right. Go ahead, click it. I dare you.

You'll be presented with a screen with an alarming number of inputs. Take some time to look through all of these, and write a good commit message. You can now choose to commit your changes and push them to Git straight away, or you can just commit your changes and start working on a new commit.

When you decide you're ready to push your commits to the server, you could go through the annoying menus, trying to find the right option. But, we promised quick and easy and keyboard shortcuts can make a lot of things much quicker. Control - Shift - K should bring up the "push changes" dialog, and it should be pretty straightforward from there!

Collaborating

If you've been working on your project with others, they might have made changes at the same time as you. As we've learned, the best way to handle this is to always pull before you push. Remember those two little arrows we found earlier? We used the green 'up' arrow for commits, and we use the blue 'down' arrow for pulling other people's changes!

That's it from us for using Git in IntelliJ - it's a very basic set of skills ready to get you started, but we strongly recommend looking at JetBrains' extremely informative guide. It'll tell you how to do pretty much anything Git-related in IntelliJ.

Good Practices

General things to keep in mind for your sanity and the sanity of your collaborators:

- Try to make commits 'atomic' - so they only do one 'thing'. It can be pretty hard to determine what 'one thing' is in a lot of cases, but try to make it so that someone else reading your code would find your commits easy to follow.
- Keep your commit messages short and sweet - less than 50 characters is best.
- Just like commits, each branch should only do one thing, but that thing can have greater scope. Generally, a branch should encompass one feature added to your project
- If you can, try to learn to rebase. We don't have the time or scope to cover it here, but rebasing is like doing Git time-travel. It's awesome.
- Learn as much as you can! Git can do so, so much more than we've talked about and a lot of it can be pretty weird.

That's all folks!

Thanks for coming to the workshop today! We hope you had a great time.

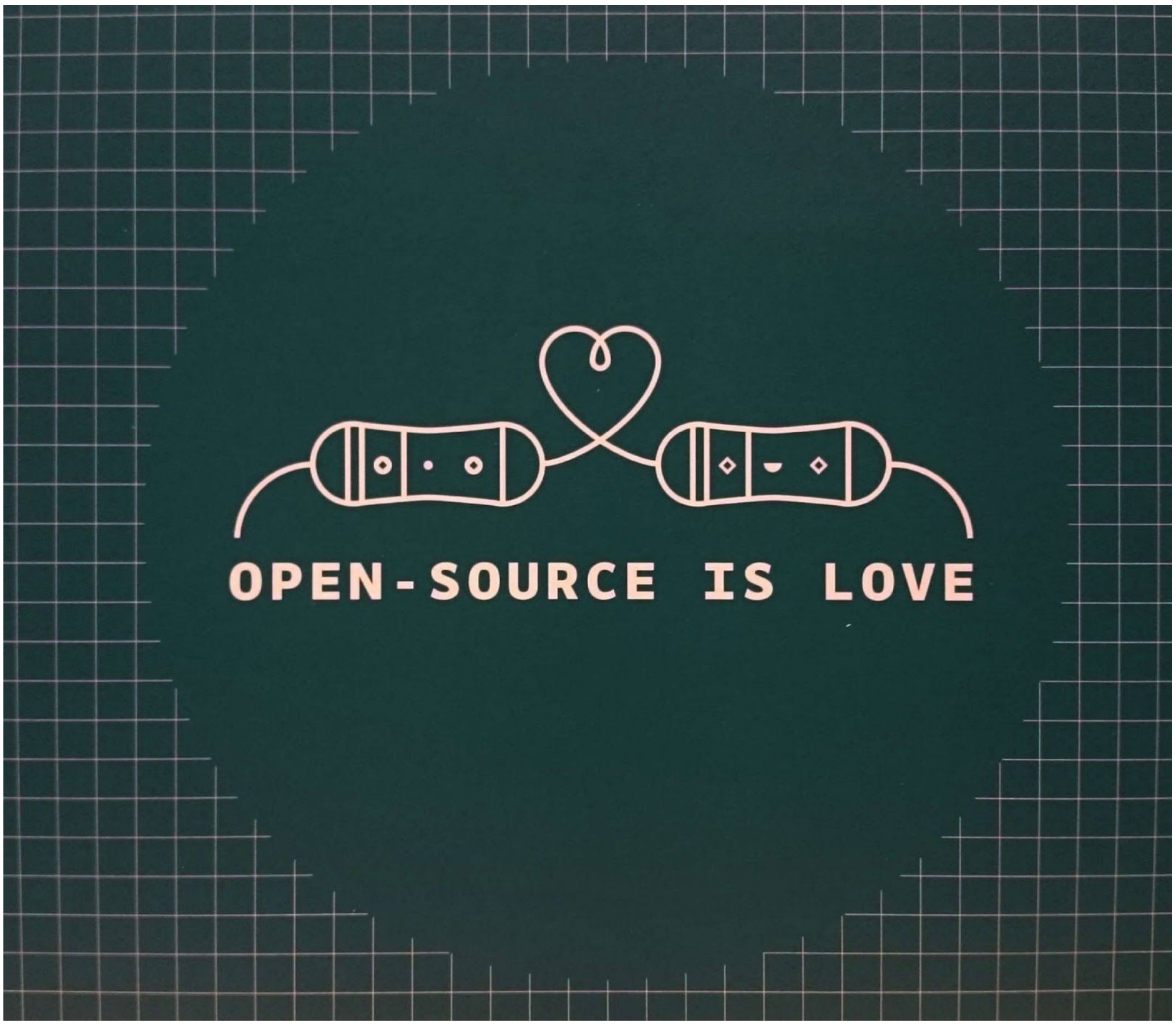
If you want to keep learning about this stuff and are wondering how, here are a couple of ideas:

- Try searching for an interesting open-source repository. There's a huge variety out there, like this Pokemon Go bot, or this big list of NASA projects. Even the full source code for the Apollo 11 Guidance Computer is on GitHub.
- For technical guidance and tutorials, have a poke through the official Git documentation (https://git-scm.com/documentation) - it's full of information, all laid out in a beautifully clear way. For GitHub-specific help, check out the GitHub guides (https://guides.github.com).

Before you go, I want to direct you to Robertson's Choose Your Own Adventure document which has gotten me out of trouble numerous times. Happy coding and collaborating from CompSoc.

Feedback

Please take a couple of minutes to fill out this feedback form. Let us know what worked and what didn't. And, feel free to let us know what you'd love to see in future events.


OPEN-SOURCE IS LOVE