Introduction to Git and Github

1 Introduction

In this short course, we'll be using the version control program 'git'. You'll learn what version control is, how to build and update a repository and how to use GitHub.

1.1 What is version control?

If you were to look at some of my old projects on my computer, you'd see a few things:

- Everything is inside a single folder, with little structure
- There are filenames like "ArticleDraft7_B_EDIT_THIS_ONE_(Backup).doc"
- There are filenames like "DataFromBob_fromEmail EDITED.xls" which had been emailed back and forth four times
- The draft with the highest number is not the one that was edited last
- There are R scripts that point to files that no longer exist on my system
- Some of the functionality relies on me remembering not to run particular lines in a script

In other words, its *disorganised*. If I or someone else wanted to pick up the project again, it would be very difficult to figure out where to start. It's also very redundant - I'm storing whole copies of files with just a few differences between them.

The answer to this mess is version control. The idea is that you store *changes* to files in a way that you can go back to the way a file was at a preivous time. It also makes it easy for different people to work on different bits of the project at the same time. By comparing the differences between people's versions, it's possible to merge changes in an organised way.

Using version control means that you can have a single file for the most up-to-date version of your article manuscript, but still keep the entire history of edits. It also basically forces you to make sure your project is well organised and documented.

There are lots of examples of programs that do version control. In this tutorial, we'll be using the program **git**, but programs like Mac's Time Machine or Google Docs or WriteLatex/Overleaf also use version control.

1.1.1 Basic concepts of version control

Git stores a **repository** of information about your project. You select files for the repository to **track**, then edit the file as usual. When you've made a significant change, you can **commit** the changes. At this point, all the differences between the last commit and the current file will be stored in the repository. That means you can **checkout** different versions of the file at different points in its history.

The general cycle of using version control is the following:

- Initialise a repository to track changes
- Add files to track
- Commit changes to the repository
- Make changes
- Commit changes to the repository …
- Make changes …
- Push repository to an external host

Pretty much all the code you'll learn in this tutorial is here:

```
> git init
> git add *
> git commit "first commit"
> git add *
> git commit "changed title"
> git push
```

1.1.2 WARNING

Version control is not the same as backing up data! Git will store the history of your project, but if your computer is damaged or lost then you also lose your project. Backing up to another hard drive / cloud storage is also necessary

1.2 Basic setup for this course

Before the course, you need to do 3 things: install git, install a modern text editor and get a free github account. The basic intro in this document will show you how to do this, and also show how to navigate directories at the command line. Finally, we'll set up some files that we'll use in the rest of the tutorial.

1.2.1 Installing git

You can download git here: https://git-scm.com/downloads

If you're on Windows, just run the installation program and select all the default settings in the install wizard. You should end up with a program called GitBash on your machine.

If you are familiar with the command line or have a mac, the link above will work fine, but there are also some more details about other ways to install git here). (I use git through the Mac Terminal, and installed it from the command line).

For the moment, I would recommend avoiding GitHub desktop.

1.2.2 Installing a modern text editor

You should also install a modern text editor, such as:

Windows: Notepad ++

Mac: [Text Wrangler] (http://www.barebones.com/products/textwrangler/download.html) or [Sublime] (https://www.sublimetext.com/)

Most Linux distributions come with a decent text editor

Note that Notepad for Windows or TextEdit for Mac can corrupt text files.

1.2.3 Getting a github account

You should sign up for a github account. It's free. Go to: https://github.com/ and click 'Sign up'.

1.3 Make some folders and files

Make a folder for this Introduction. Inside that folder, make a folder called `tutorial1`.

Make a new text file called `animals.txt` with the following text:

```
Donkey
Dolphin
Monkey
Baracuda

```

Note the blank last line

Save `animals.txt` to the `tutorial1` folder.

1.4 Navigating directories in the terminal (*cd*, *pwd* and *ls*)

This section introduces basic navigation commands such as `cd`, `pwd` and `ls`. If you're familiar with these, you can skip to the next section.

Start up your terminal (Mac/Linux) or GitBash (Windows). These use standard Linux bash commands.

This is where you can type commands to run programs like git. Be careful - the terminal is very powerful and some commands can delete files. But if you follow this tutorial, you shouldn't run into any problems.

At any point, the commands you type will apply to the **working directory** ("directory" means the same thing as "folder"). To find out what the current directory is, type `pwd` (print working directory). The box below shows the command you should type. The `>` symbol represents the command prompt, and you don't have to type this character. Press enter after each line. Any line that does not have a command prompt in front of it is output - you shouldn't type this.

```
> pwd
```

Type this and press enter. For me, the pwd command returns `/Users/sgroberts`. For you, it will probably be your home directory. We want to set the current directory to the `tutorial1` folder. You can **change directory** using the `cd` command, followed by the location of the directory you want to change to. On my computer, the `tutorial1` folder is here:

```
> cd ~/Documents/Teaching/IntroToGitHub/TutorialFolders/tutorial1/
```

The `~` character at the start is a shortcut for "my home directory".

Now we can check the working directory again:

```
> pwd
/Users/sgroberts/Documents/Teaching/IntroToGitHub/TutorialFolders/tutorial1
```

If we're navigating just one folder up or down, then there's no need to type the whole thing. To move one directory up, type:

```
> cd ..
```

And to move into a sub-directory of the working directory, just use the name of the directory

```
> cd tutorial1
```

You can get a **list of files and folders** inside a directory by using `ls`:

```
> ls
animals.txt
```

The `tutorial1` folder has 1 file inside called `animals.txt`.

1.4.1 Go on to the next section

Introduction to Git and Github: Tutorial 1 Basics of git

1 Introduction

In this tutorial, we'll learn the basics of the program git:

- Initialise a repository
- Adding files to track
- Making and committing changes
- Undoing and redoing (checkout and revert)

1.1 Initialise a git repository

After navigating to the `tutorial1` folder, let's initialise a repository, type `git init` and press enter:

```
> git init
Initialized empty Git repository in /Users/sgroberts/Documents/Teaching
  /IntroToGitHub/TutorialFolders/tutorial1/.git/
```

This created a hidden folder in our directory called `.git`. You may not be able to see it in your file browser, but that's ok - we don't need to look inside for now.

Let's check the status of the repository:

```
> git status
```

You should get something like this:

```
On branch master

Initial commit

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

    animals.txt

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

Git is telling us that we're on the master branch, labelled 'Initial commit'. It also says that there are untracked files in our

folder. That means that there are files that the repository is not tracking yet.

1.2 Adding files to the repository

Let's add the file `animals.txt` to the repository:

```
> git add animals.txt
```

Now we can look at the status again:

```
> git status

On branch master

Initial commit

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

    <span style="color:red">new file:   animals.txt</span>
```

It says that there's a new file waiting to be commited.

1.3 Commiting changes

Let's commit this file!

A commit is done with the git `commit` option. Every time we commit we need to add a **commit message** about the changes we made. This will come in useful later. The message can be anything you like, but should make it clear what the changes you made are.

To add a commit message, use the `-m` option, then surround a short message surrounded by double quotes:

```
> git commit -m "Added animals.txt"

[master (root-commit) 3a397b1] Added animals.txt
 1 file changed, 4 insertions(+)
 create mode 100644 animals.txt
```

The message tells us that the commit contained a change to 1 file with 4 insertions. Git tracks changes to each line of each file. The 4 insertions are the 4 lines of text inside `animals.txt`.

1.3.1 Making changes

Let's edit the `animals.txt` file to the following:

```
Donkey
Shark
Dolphin
Barracuda
Jellyfish
```

Save the text file.

Note that we fixed the spelling of "Barracuda", deleted *Donkey* and added *Shark* and *Jellyfish*. If we run `git status` again, git shows us the files that have changed since the last commit:

```
> 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:   animals.txt

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

We can also look at more detailed information on what has changed since the last commit with `git diff`

```
> git diff

diff --git a/animals.txt b/animals.txt
index c7896ad..b35d465 100644
--- a/animals.txt
+++ b/animals.txt
@@ -1,4 +1,5 @@
 Donkey
+Shark
 Dolphin
-Monkey
-Baracuda
+Barracuda
+Jellyfish
```

This tells us that there have been chagnes to the file `animals.txt`. Additions are coloured in green and deletions are coloured in red. Note that the editing of “Baracuda” has been counted as a deletion and an addition.

1.3.2 Committing changes again

We haven’t added the changes yet, so let’s do that:

```
> git add animals.txt
```

This command tells git to add all changes in all files inside the working directory (and all files in sub-directories inside the working directory).

Let’s commit these changes:

```
> git commit -m "Fixed Barracuda spelling, added shark and jellyfish, deleted monkey"

[master bc7f5ac] Fixed Barracuda spelling, added shark and jellyfish, deleted monkey
 1 file changed, 3 insertions(+), 2 deletions(-)
```

Git tells us that 1 file has changed, and we’ve made 3 insertions and 2 deletions.

The command `git log` shows us the history of commits, starting with the most recent commit:

```
> git log

commit bc7f5ac9be1fe3dc8a4a779d81e17bf5f6bb7962
Author: seannyD <sean.g.roberts@gmail.com>
Date:   Sun Jul 3 12:15:42 2016 +0200

    Fixed Barracuda spelling, added shark and jellyfish, deleted monkey

commit 3a397b1bc84f63f149a87f893e5013090f65968b
Author: seannyD <sean.g.roberts@gmail.com>
Date:   Sun Jul 3 11:59:25 2016 +0200

    Added animals.txt
```

1.4 Undo and Redo

We'd like to undo the changes made to `animals.txt` since the last commit. The first step is to get the ID of the commit we want to go back to. Use `git log --online` to show a list of commits with just the IDs and commit messages:

```
> git log --oneline

981cd55 Fixed Barracuda spelling, added shark and jellyfish, deleted monkey
6fec032 Added animals.txt
```

The line starts with an ID code for the commit, and the message we entered for the commit. This is why the commit messages are important. It's tempting to be vague in the message, but documenting changes makes it easier to do stuff later on!

From xkcd https://xkcd.com/1296/

We want to go back to commit `6fec032`.

To undo changes to a file since the last commit, you can use the `checkout` command. It takes two main arguments - the ID of the commit, and what file you want to checkout.

```
> git checkout 6fec032 animals.txt
```

Re-open the `animals.txt` file again. You should see that the contents of the file have been changed back to how it was at the first commit. You can now edit this file, and then add these chagnes to the repository and make new commits, just like before.

You can change the file back to the most recent commit ("redo") by checking out the HEAD:

```
> git checkout HEAD animals.txt
```

The file should now have 5 animals again.

1.4.1 Tracking multiple files

Let's make another text file inside the `tutorial1` folder called `plants.txt` with the following text:

```
oak
daffodil
```

We could add this file to the repository with `git add plants.txt`, but we can also add *all* files in the directory like this:

```
> git add *
```

This would also add *all files in sub-directories*. Let's commit the changes:

```
> git commit -m "Added plants.txt"

[master 1d8d88c] Added plants.txt
 1 file changed, 2 insertions(+)
 create mode 100644 plants.txt
```

1.4.2 Checking out all files

We can now use `checkout` to undo *all* files back to a previous commit.

First, let's find the ID:

```
>git log --oneline

1d8d88c Added plants.txt
981cd55 Fixed Barracuda spelling, added shark and jellyfish, deleted monkey
6fec032 Added animals.txt
```

Let's go back to the commit before we added plants.txt:

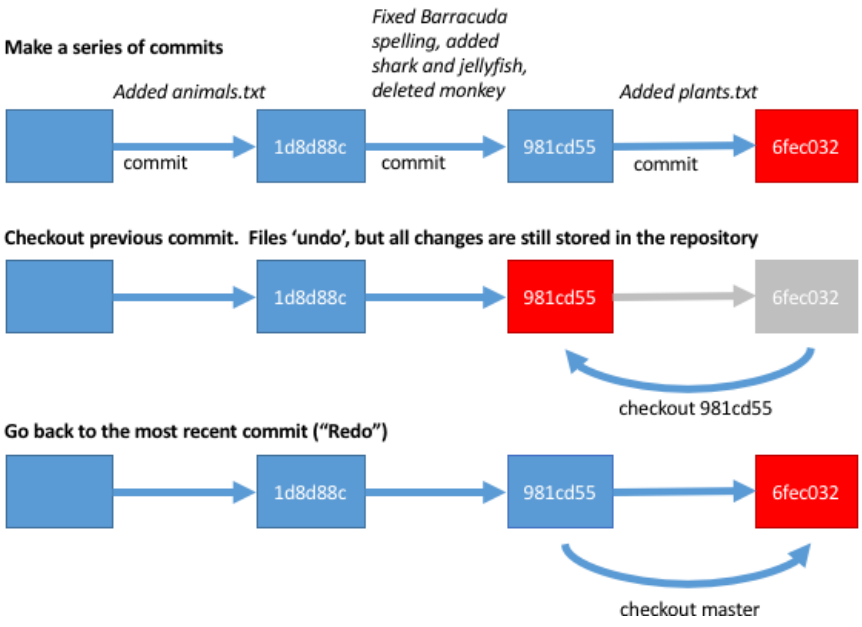
```
> git checkout 981cd55

Note: checking out '981cd55'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 981cd55... Fixed Barracuda spelling, added shark and jellyfish, deleted mo
```

Now the folder will go back to how it looked before - we only have one file `animals.txt`.

We get quite a long message warning us that we're in a 'detached HEAD' state. This means that you can make any changes you want to the files in your directory and also make commits of these changes, but you can always return all files to the state that they were in.

```
> git checkout master
```

Here's a diagram of what we did:



One good use for academia is to make a commit when you submit a project to a journal or conference. Mark this as e.g. “Version submitted to Nature”. When you get revisions back from reviewers, you can check what the project looked like when you submitted.

1.4.3 Reverting to a previous state

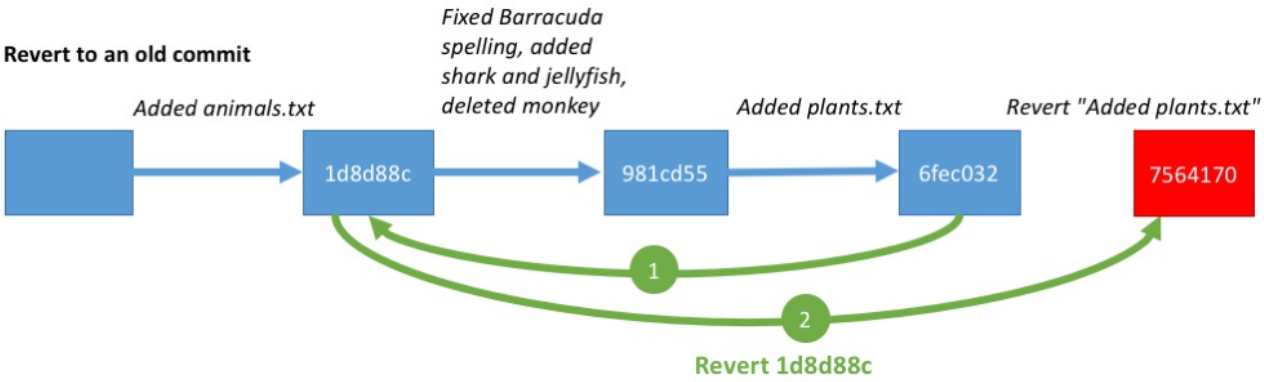
The checkout command lets us “check out” the state of files at a previous time. But if we want to actually want to reverse the changes and keep working on the files as they were before, then we need to “revert”. Let’s revert back to the state after we added `animals.txt`.

```
> git revert 1d8d88c
```

You’ll be asked for a revert message. The default is fine.

Note that on some systems, you’ll be asked to write the revert message in a terminal-based editor like vim. Usually you can type `:` then `q` to continue.

This command actually ‘undoes’ everything back to the commit we specified, but then works out the changes between that and the current commit, and adds a *new* commit at the end.



There’s now an extra commit in the log. Doing it this way means that we can undo changes, but still go back to the way the project looked after we fixed the barracuda spelling etc., if we want.

1.5 Review

In this tutorial, we learned how to initialise a repository: add files and commit changes:

```
> git init
```

Then we make changes, add changes to the repository and commit the changes to the repository:

```
> git add *
> git commit -m "Put a message here to describe the changes"
```

The two commands above are the ones you’ll use the most.

We can also look at a list of commits to find commit ids:

```
> git log --oneline
```

And go back to a previous state of a file:

```
> git checkout <commit id> fileName.txt
```

Or ‘revert’ back to a previous state of the whole project:

```
> git revert <commit id>
```

1.5.1 Go on to the next section

Introduction to Git and Github: Tutorial 2 Pushing a repository to GitHub

1 Back to Tutorial 1

2 Introduction

In this tutorial, we'll learn how to link a repository to GitHub.

To start, open your terminal / GitBash and **navigate to the folder you made in tutorial 1**.

2.1 WARNINGS

GitHub repositories are **public** by default. That means that anything you upload to GitHub can be seen by others. It also means that other people can see **any data that exists in your commit history**. This can include old drafts of papers or data before it was anonymised.

If you pay for a github membership, you can create private repositories, or you can use other services. e.g.:

- Apache Subversion (svm). There's a local, secure repository available from the Nijmegen MPI TG,svn.mpi.nl.
- Gitlab hosting service, available through MPG (https://gitlab.gwdg.de).

See the last tutorial for a way of making sure some files are not included in the repository.

2.2 Set your git username

When collaborating, it's good to know who makes what changes. You can tell git what your github username and email address is using `git config`.

My github name is seannyD and my email is sean.roberts@hotmail.com, so I would use:

```
> git config --global user.name "seannyD"
> git config --global user.email "sean.roberts@hotmail.com"
```

Set your own username and email now.

Note that the "–global" command sets your username for all repositories. You can set the username just for the current repository by navigating to the repository and using e.g. `git config user.name "seannyD"`.

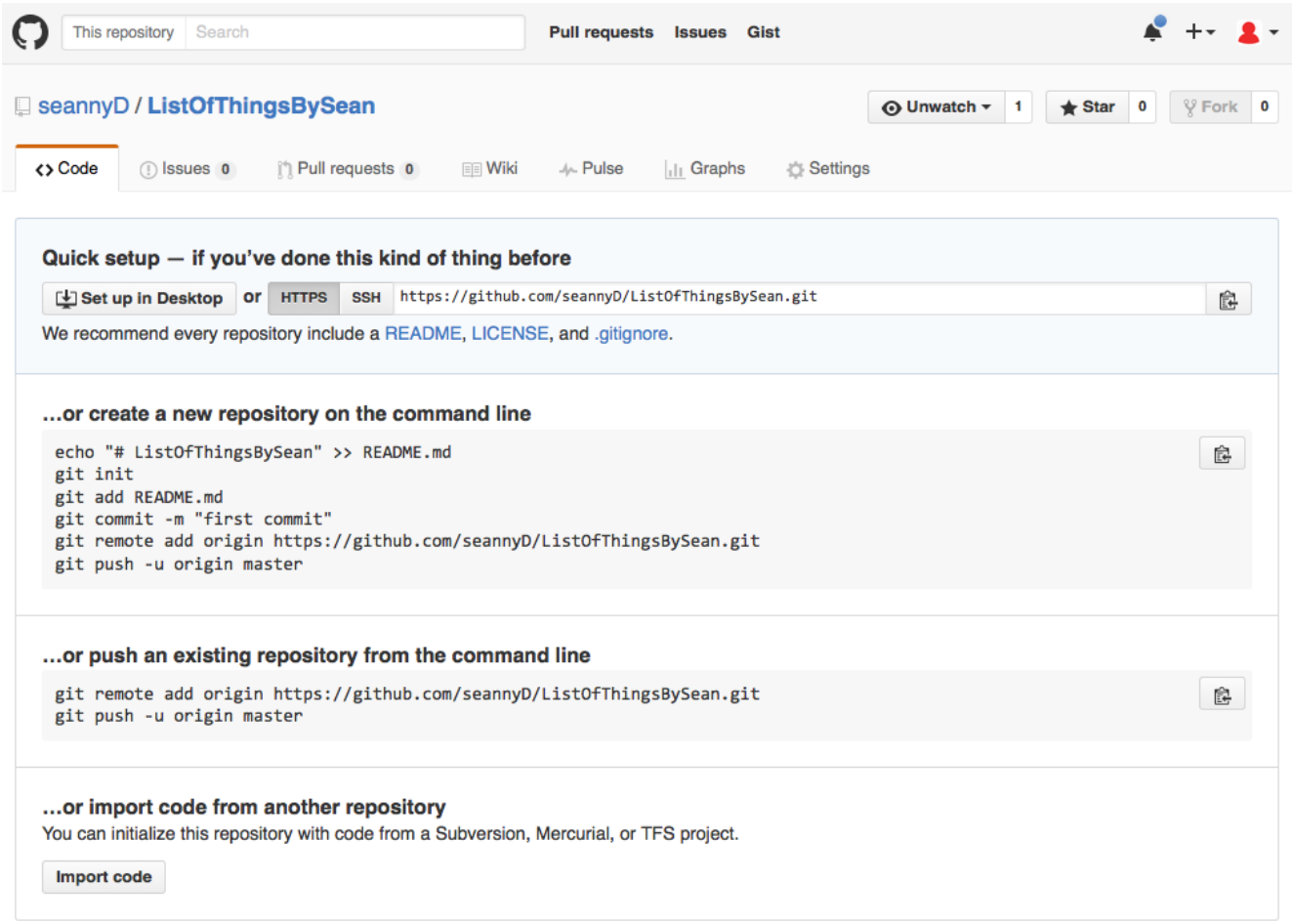
2.3 Adding a remote repository

In tutorial 1 we created a git repository. Let's link that to an online repository on GitHub.com.

- Go to www.github.com and log in.
- Click 'New repository'

- Choose a name for the repository. This must be unique for all of GitHub, so make it obvious. For this tutorial, make it something like “ListOfThingsBySean”.
- Click ‘Create repository’ (the default options are fine, and you can add a description later)

You’ll get a page like this:



Currently, the repository is empty, so it gives you four options:

- quick setup deatils (if you know what you’re doing)
- create a new repository on the command line
- push an existing repository from the command line
- import code from another repository

It also gives some code. Note that the second option has some familiar commands: git init, git add and git commit.

But we already have a repository, so we want the third option: “push an existing repository from the command line”. Copy the two lines of code and paste them into your terminal / GitBash. My code looks like this, but you’ll need to **replace the web address** with your own repository address.

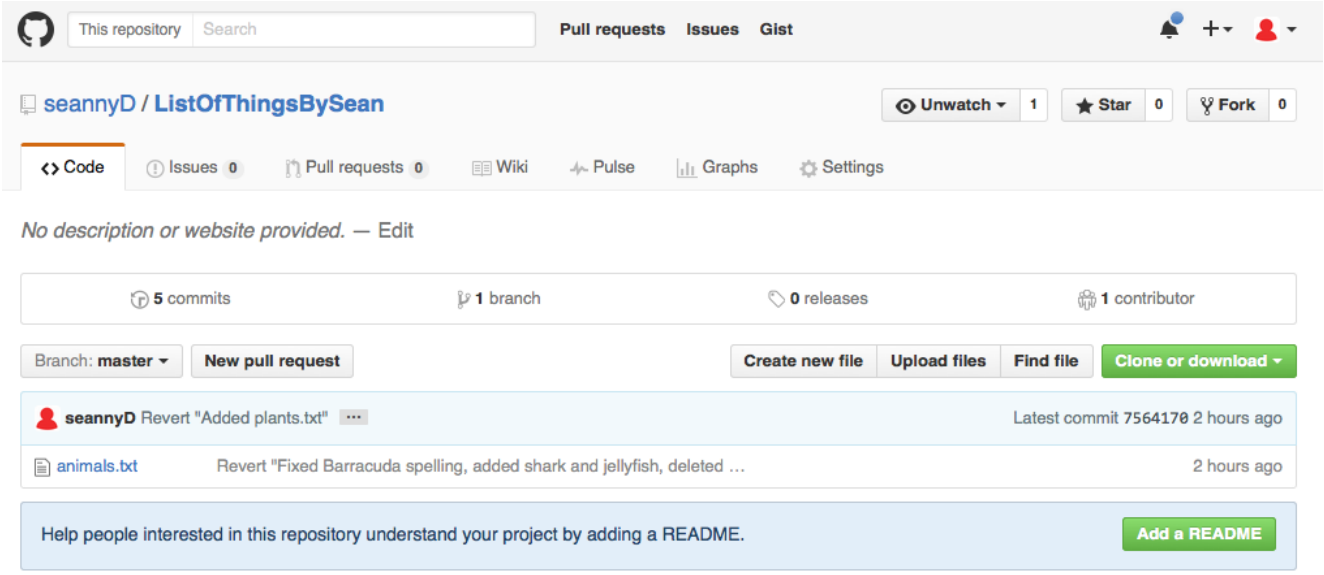
```
> git remote add origin https://github.com/seannyD/ListOfThingsBySean.git
> git push -u origin master
```

The first command tells git that you’re going to link the repository to the given web address.

The second command tells git to upload the repository to GitHub. The `-u origin master` tells git to push the master branch to GitHub, and to remember this option for later. From now on, you can just use `git push` to send things to GitHub.

You’ll need to enter your GitHub password. There are a number of ways of avoiding doing this every time, see here.

After the files have uploaded, go back to the GitHub page in your browser and refresh the page. You’ll see something like this:



The files in your directory are now copied to the GitHub page, and other people can access it. It will updated every time you make a commmit and push.

There are lots of options things on this page, but one of the most useful for now is the ability to add collaborators - people who can edit and push to your online GitHub repository. To do this, go to *Settings* > *Collaborators* and enter the username/email address of collaborators.

2.4 Review

We've now learned the basics of git:

Initialise a repository

```
> git init
```

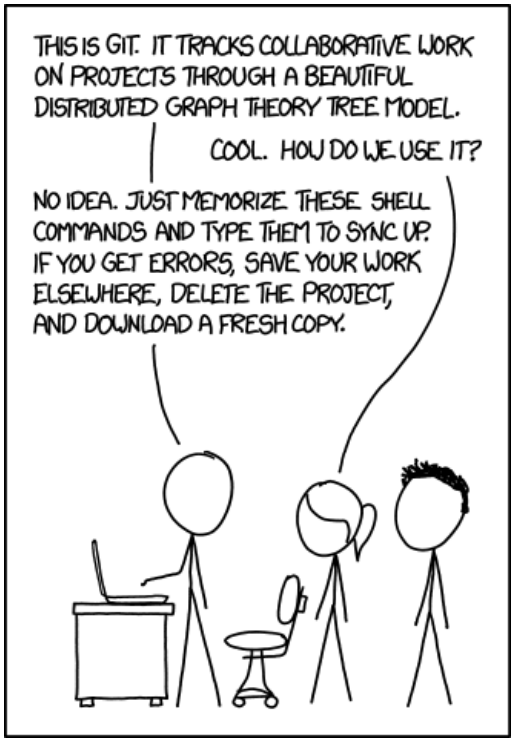
Make a GitHub repository on GitHub.com, then link your local repository to it:

```
> git remote add origin <repository url>
> git push -u origin master
```

You'll now be mainly using these three commands every time you make changes:

```
> git add *
> git commit -m "Commit description message"
> git push
```

You can now appreciate this comic:



From xkcd, https://xkcd.com/1597/

From xkcd https://xkcd.com/1597/

2.4.1 Go on to the next section

Introduction to Git and Github: Tutorial 3 Collaboration using GitHub

1 Back to Tutorial 1

2 Back to Tutorial 2

3 Introduction

In this tutorial, we'll learn how to collaborate on projects using git and GitHub.

To do this tutorial, you'll need to be added as a collaborator on one of my GitHub projects. If you haven't sent me your GitHub username, please email me.

3.1 Designing projects for collaboration

3.1.1 File structure

It's a good idea to have a very clear file structure for your project. This means that it's easy to locate files, and it's obvious where new files or changes should go.

I usually have something like this:

- Main folder (Top folder for the version control)
- data

    - Raw data
    - Processed data
- processing (any scripts for processing raw data)
    - R
    - Python
- analysis (any scripts for analysing results)
    - R
- results (for storing results)
    - graphs
- writeup (for storing drafts of articles)

3.1.2 Reproducability

An ideal GitHub project will be completely self-sufficient and reproducible. That is, someone else should be able to download just the files that are in the repository, and reproduce all your results and output. To achieve this, consider

- Including all raw data inside the repository
- Including all scripts that process the data and produce graphs/results
- Using relative file paths in scripts. This means you should use paths like "../../data/RawData/Part1.csv", rather than "C:/MyDocuments/Sean/LingProjects/data/RawData/Part1.csv". This means that the scripts should run on somebody else's computer.

3.1.3 Documentation

Documentation is important. It means having a "README" file for the project / each subfIder, describing the contents of each file. If you include a file named "README.md" in the top directory of your repository, GitHub will use this as the project's main page on the web. You should also document your code, describing what each function and set of lines does.

3.2 Collaboration on a GitHub project

I made a GitHub project here:

https://github.com/seannyD/SeansGitHubTutorial-Collaboration

Go to this page and have a look at the files and folders. It's a project for a timed list experiment. Participants have to list as many items from a semantic domain as possible. The responses are written to a file, one line per response. Like this:

```
cow
pig
sheep
salmon
```

The raw files can be found in the folder data/rawData/animals.

The name of the file is the participant name/id followed by an underscore, then either 'm' or 'b', indicating if the participant was brought up monolingually or bilingually.

There is an R script for collating and analysing the data in `analysis/R/analyseData.Rmd`, and this writes the results to `results/MainResults.html`.

3.2.1 Cloning a repository

We'd like to add your own resonses to the project. The first step is to clone the repository from GitHub to your own local machine.

Make another folder for this collaboration tutorial. Make sure it is **not inside the folder for the previous tutorial**, or inside any other folder which already has a github repository.

Navigate to this folder in your terminal / GitBash. For me it's something like:

```
> cd ~/Documents/Teaching/IntroToGitHub/TutorialFolders/collaborationTutorial
```

Next we need to find the URL of the repository. Go to my repository page:

https://github.com/seannyD/SeansGitHubTutorial-Collaboration

Look for a button labelled 'Clone or download'. Click it and a little box will appear with a link. Copy this link.

The link should be:

```
https://github.com/seannyD/SeansGitHubTutorial-Collaboration.git
```

Note that this is just the url of the main page for the project, but with ".git" at the end.

In your terminal, type

```
> git clone https://github.com/seannyD/SeansGitHubTutorial-Collaboration.git
```

You should see output like this:

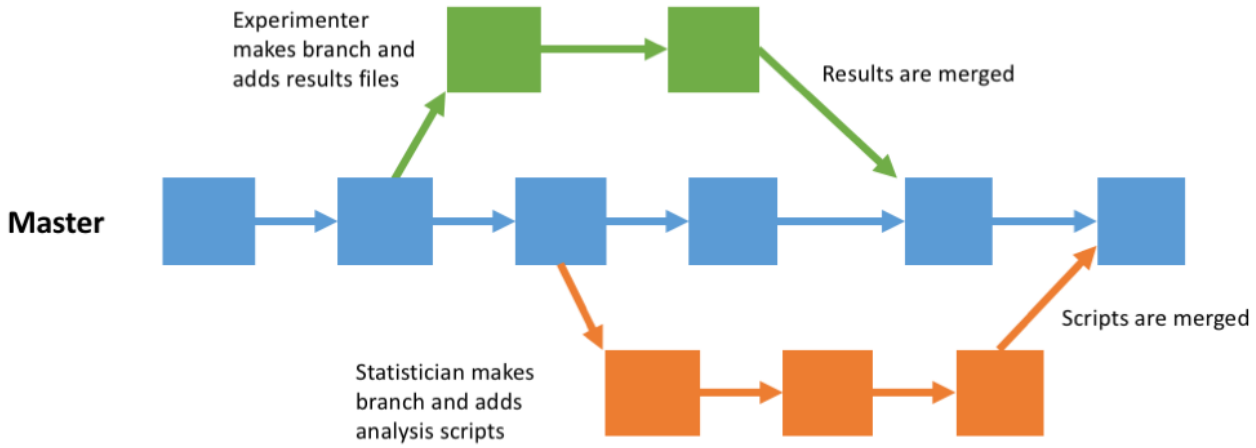
```
Cloning into 'SeansGitHubTutorial-Collaboration'...
remote: Counting objects: 26, done.
remote: Compressing objects: 100% (20/20), done.
remote: Total 26 (delta 3), reused 26 (delta 3), pack-reused 0
Unpacking objects: 100% (26/26), done.
Checking connectivity... done.
```

Git should now download both the files and the repository structure to your machine. You should have a folder inside your working directory named `SeansGitHubTutorial-Collaboration`.

3.3 Branching

We're now going to edit our local version of the project, then upload our edits to the main project. Because other people might be working on this project at the same time, we want to work on a **branch**.

The idea is that people can work on their own branches independently, then **merge** the branches into the master branch later.



Now that we've cloned the repository, let's navigate inside the project folder. You should type something like:

```
> cd SeansGitHubTutorial-Collaboration
```

Let's check which branch we're on:

```
> git branch
* master
```

This repository only has one brach - the master branch - and that's our currently checked-out branch (you can tell by the * symbol).

Now we can make our own branch to work on. I'm going to call my branch "SeanAddData", but you should name it something so that it's clear what's going on in this branch.

```
> git branch SeanAddData
```

Let's check the branches again:

```
> git branch
    SeanAddData
  * master
```

There are now two branches, but the master banch is still the checked-out one.

Let's checkout our branch (remember, your branch name will be different):

```
> git checkout SeanAddData
Switched to branch 'SeanAddData'
```

3.4 Making changes

We can now make changes as before.

Add your own responses file to data/rawData/animals/. Make sure there's one response per line, and that you name file something like "YourName_m.txt" (where m is monolingual and b is bilingual).

Add your changes to the repository branch

```
> git add *
```

Commit your changes, adding a commit message that makes sense:

```
> git commit -m "Added data from <Your name>"
```

3.4.0.1 Pushing the branch back to GitHub

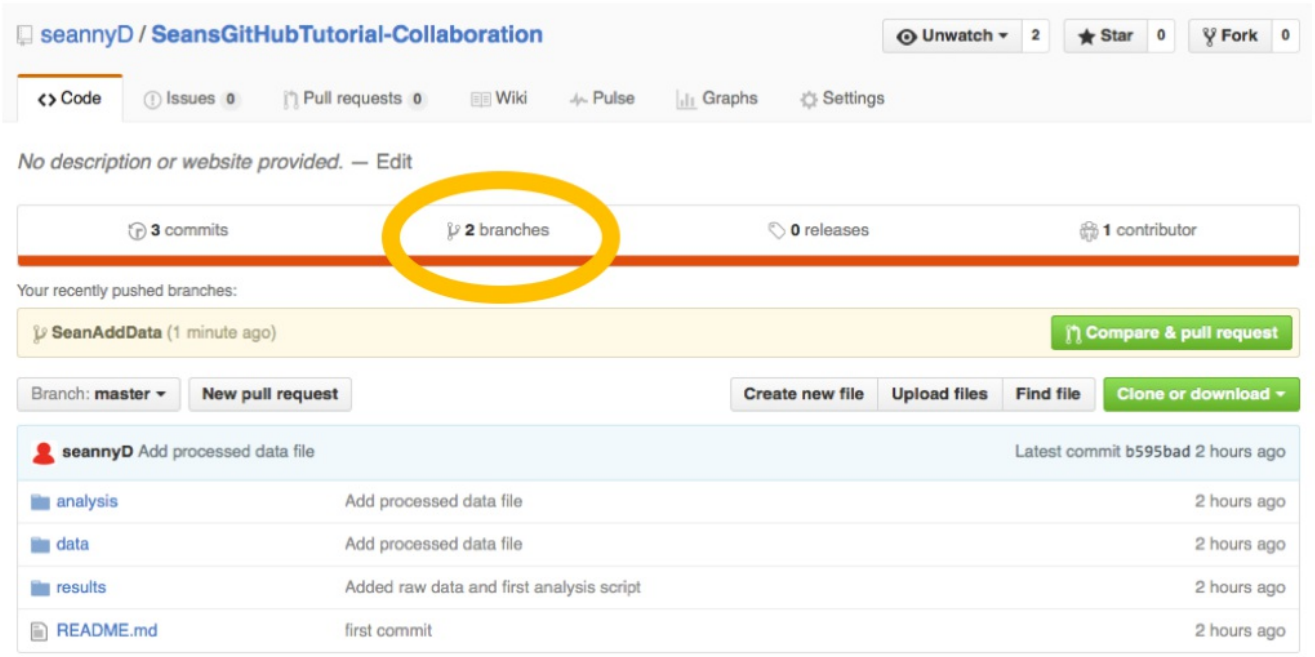
Push your branch to the GitHub repository. The first time we do this, we need to tell git that we want the 'upstream' branch to be the GitHub branch (the 'origin').

```
> git push --set-upstream origin <Your branch name>
```

You should get output like this:

```
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 647 bytes | 0 bytes/s, done.
Total 6 (delta 1), reused 0 (delta 0)
To https://github.com/seannyD/SeansGitHubTutorial-Collaboration.git
 * [new branch]      SeanAddData -> SeanAddData
Branch SeanAddData set up to track remote branch SeanAddData from origin.
```

If we go back to the GitHub website, we now see that the project has 2 branches:



3.5 Merging branches

The owner of the GitHub project can actually compare and merge branches directly on GitHub. There are many powerful features such as allowing people who have created branches to request that the owner merge their branch, and a discussion forum for debating how merges should happen.

However, for now we'll just show how to merge branches on the command line.

In what follows, I assume that a collaborator (you):

- cloned the repository
- made a branch
- commited changes to the branch

- pushed the changes to the main GitHub page

In the next section, the owner (me) will:

- pull this change to their own local repository
- merge the branches
- and then push the updated branch back to GitHub.

3.5.0.1 Pull the most recent changes

The first step is to make sure that your repository is up to date. Others may have made changes since you created your branch. You can pull changes at any time, even if you don't intend to merge branches.

First, let's make sure we've switched back to the master branch:

```
> git branch master
```

Now let's **pull** new data from GitHub (this is the opposite of**push**).

```
> git pull

remote: Counting objects: 6, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 1), reused 6 (delta 1), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/seannyD/SeansGitHubTutorial-Collaboration
 * [new branch]      SeanAddData -> origin/SeanAddData
Already up-to-date.
```

3.5.0.2 Merging branches

Let's check the branches we have available. Note that we're using the `-a` option to make sure we see local and remote branches.

```
>   git branch -a
* master
  remotes/origin/SeanAddData
  remotes/origin/master
```

Our checked-out branch is the master branch, and can see the new branch (origin/SeanAddData). Let's merge it in using the `merge` command:

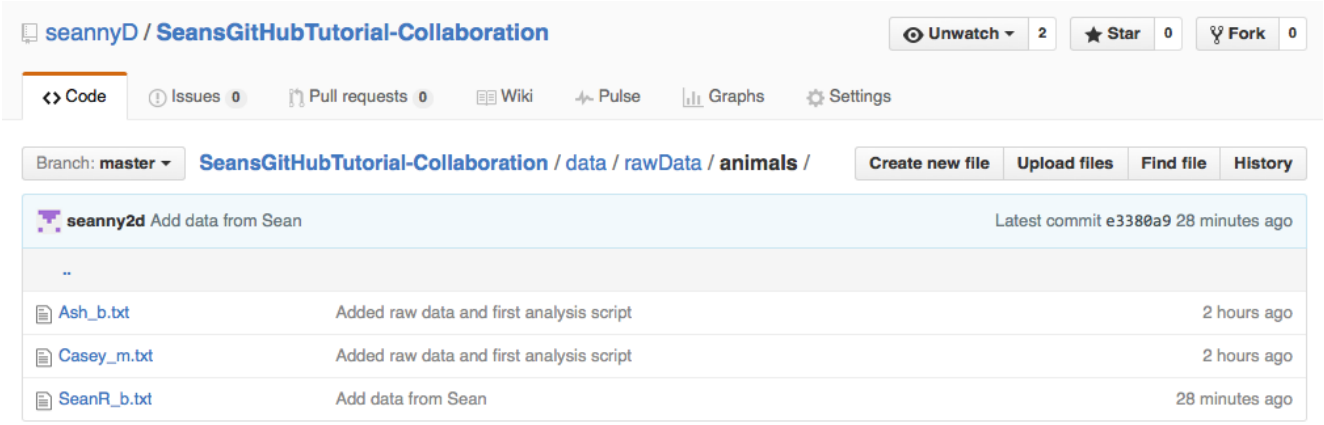
```
> git merge origin/SeanAddData

Updating b595bad..e3380a9
Fast-forward
 data/rawData/animals/SeanR_b.txt | 9 +++++++++
 1 file changed, 9 insertions(+)
 create mode 100644 data/rawData/animals/SeanR_b.txt
```

We get a message saying that 1 file was added to the master branch.

And let's push the changes to GitHub:

```
> git push
```

On the website, we can see that the rawData folder has a new file (SeanR_b.txt) in it, with the commit message that we made in our branch.



3.6 Viewing other people's changes

When you pull updates from GitHub, you'll probably want to know what's changed. First, let's see what the commit messages are like:

```
> git log --oneline

2a06936 edited readme
e3380a9 Add data from Sean
b595bad Add processed data file
d4d7a4f Added raw data and first analysis script
26bf47a first commit
```

Let's say we want to find out what happened between "Add data from Sean" and "edited readme". We can use `git diff` followed by the two commit IDs we want to compare.

```
> git diff e3380a9 2a06936
```

This gives the following output:

```
diff --git a/README.md b/README.md
index 5d4f2d8..9a02461 100644
--- a/README.md
+++ b/README.md
@@ -1 +1,9 @@
-# Collaboration tutorial# SeansGitHubTutorial-Collaboration
+# Collaboration tutorial
+
+It's a project for a timed list experiment.  Participants have to
 list as many items from a semantic domain as possible.  The
 responses are written to a file, one line per response.
+
+The raw files can be found in the folder [data/rawData/animals]
(https://github.com/seannyD/SeansGitHubTutorial-Collaboration/tree/master/data/rawData/a
+
+The name of the file is the participant name/id followed by an
underscore, then either 'm' or 'b', indicating if the participant was
 brought up monolingually or bilingually.
+
+There is an R script for collating and analysing the data in
`analysis/R/analyseData.Rmd`, and this writes the results to
`results/MainResults.html`.

diff --git a/data/.DS_Store b/data/.DS_Store
index fc68288..a3d5c16 100644
Binary files a/data/.DS_Store and b/data/.DS_Store differ
diff --git a/data/rawData/.DS_Store b/data/rawData/.DS_Store
index 5008ddf..60cf046 100644
Binary files a/data/rawData/.DS_Store and b/data/rawData/.DS_Store differ
```

This means that there were edits to **README.md**. The line at the start of the file "# Collaboration tutorial#

SeansGitHubTutorial-Collaboration", was deleted and several other lines were added. Basically, the last edit just updated the README file to include a fuller description.

3.7 Merging branches in your local project

In the example above, the collaborators made branches, then the project owner merged them. But you can merge branches in your own local repository. You can use `merge` in the same way as above:

Make sure you're in the branch you want to merge*into* (e.g. the master branch:

```
> git branch master
```

Then merge the branches

```
> git merge NameOfBranchToMerge
```

3.8 Review

In this tutorial we learned how to view branches:

```
> git branch
```

… create new branches:

```
> git branch newBranchName
```

… switch branches:

```
> git branch nameOfBranchToSwitchTo
```

We learned that, if you're collaborating, development should always be done on a branch, not the master. We can make changes to our local branch with `git add` and `git commit` as before. We can push our branch to the GitHub page with `git push`. When we're done, we can merge branches:

```
> git merge branchToMergeFrom
```

3.8.1 Go on to the next section

Introduction to Git and Github: Tutorial 4 Miscellaneous features of git and GitHub

1 Back to Tutorial 1

2 Back to Tutorial 2

3 Back to Tutorial 3

4 Introduction

4.1 Untracking files

The `reset` command will remove the file from the repository, but it won't delete the file from your system.

```
> git reset plants.txt
```

4.2 Avoid adding specific files to a repository

Let's say that you don't want to track the changes to specific files in your repository. For instance, you might have personal information that you don't want to make public.

Add a text file to the top directory called `.gitignore` (note the dot at the start, no '.txt' at the end), and add the files you want to avoid adding on a seperate line. e.g.:

```
data/participantNames.csv
consentForms/*.pdf
```

This applies to the file "data/participantNames.csv" and all files in the "consentForms" folder ending in ".pdf". These won't be added to the repository.

But be careful! This will only stop files being **added**. If they are already in the repository, this won't remove them, so you need to add a gitigore file before your first commit.

If you've added files to a repository, but you shouldn't have, there are ways of removing them. But the safest and simplest thing to do is to copy the files to another location, delete the old repository and then make a new repository with a .gitignore file.

4.3 Forking GitHub repositories and pull requests

If you aren't a collaborator on a GitHub project, you can still contribute to it by forking the repository, making changes, then making a pull request.

First you "fork" the existing repository. This is actually just making a branch of the repository on the GitHub server. Clone this fork to your local machine with `git clone`. You can then make changes, make a commit and push the commit to your GitHub fork. Then you synch your repository, and make a "pull request". This sends a reqest to the owner of the project for them to merge your fork/branch with the master branch of the project.

There are more details here

4.4 Caching github password

You can store your password secuerly on your machine to avoid having to type it in at every puhs. There are a number of ways of doing this, see https://help.github.com/articles/caching-your-github-password-in-git/.

4.5 Viewing GitHub content directly

You can use services like htmlpreview to view html files in a repository. e.g.:

https://htmlpreview.github.io/?https://github.com/seannyD/SeansGitHubTutorial-Collaboration/blob/master/results/MainResults.html

4.6 GitHub licences

By default, GitHub projects do not specify a license. When you create a GitHub repository, there's an option to add a license from a list of candidates. All that this does is add a file called LICENCE.txt into your project with the legal terms of the licence. The suggested options limit people's ability to take and modify your work. A better option for freer distribution which allows people to copy and modify as long as they attribute you is the "GNU AGPLv3" (GNU Affero General Public License v3.0) see here fore more details. Note that most Creative Commons licences are not suitable for software. More options can be found on this site.

4.7 fetch versus pull

The command `git pull` actually does two things: it downloads stuff from GitHub, then performs a `git merge`. If you just want to download stuff, use `git fetch`.

4.8 Avoid tracking mac .DS_Store files

.DS_Store files are files that macs use to keep track of how you view folders. They're not really important to a project, but can cause conflicts because different users will modify the files in different ways. There's one of these files in every folder, so a simple line in .gitignore won't work. You can tell git to always ignore .DS_Store files in all repositories:

```
> echo .DS_Store >> ~/.gitignore_global
> git config --global core.excludesfile ~/.gitignore_global
```

Note that this is not retroactive. To remove .DS_Store files from an existing repository, you can use the following line. But be careful - this uses `git rm` which can delete files from your system.

```
> find . -name .DS_Store -print0 | xargs -0 git rm --ignore-unmatch
```