Practical 1: Getting Started

Getting to grips with Jupyter, Git and Markdown

Table of contents

1 Running Docker **2**

2 Setting Up GitHub **5**

3 Updating the .gitignore File **7**

4 Creating Your First Remote File **9**

5 Setting Up Git Locally **10**

6 Setting up a GitHub Web Site **15**

7 Other Ways of Using Docker **16**

8 Answers **16**

This week's practical is focussed on getting you set up with the tools and accounts that you'll need to across many of the CASA modules in Terms 1 and 2, and familiarising you with 'how people do data science'. Outside of academia, it's rare to find a data scientist who works entirely on their own: most code is collaborative, as is most analysis! But collaborating effectively requires tools that: get out of the way of doing 'stuff'; support teams in negotating conflicts in code; make it easy to share results; and make it easy to ensure that everyone is 'on the same page'.

You *will* find things confusing this week, but they will start to make more sense as we move further into the module. The key is to keep trying things out and to ask for help when you get stuck.

> ⚠ Windows Users
>
> Please use **Power Shell** instead of the **Command Prompt** (`cmd`). Power Shell behaves much more like the Terminal in macOS and Linux, so the results are more consistent and easier to 'debug'.

If you haven't done this already, you need to follow the instructions for installing all of the tools listed on on the CASA Computing Environment web page. This process will take time, so please keep reading or work on other things to do while you wait!

> **Tips on Using the Command Line**
>
> If you need help understanding how to use the Command Line or want to be able to do *much more* there are a wide range of tutorials available.
> Here are some starting points for learning more:
>
> - **I need help understanding**: Software Carpentries is your friend! They have an entire tutorial titled The Unix Shell.
> - **I *still* need help understanding**: the Programming Historian is another good place to look! And they *also* have an entire tutorial titled An Introduction to Bash.
> - **I want to do *more* on the Command Line**: O'Reilly has produced an online book called Data Science at the Command Line that will take you much, much further.

> **Following Convention**
>
> A common convention in programming is to use the `<some text or other>` to indicate that you should replace the text between the `<` and `>` with something that makes sense in your context. For example, if you see `cd <directory>` you should type `cd Documents` if you want to change to the `Documents` directory. If you see `<your username here>` you should type `jreades` if that's your username. And so on. You do *not* type the `<` or `>` characters!

1 Running Docker

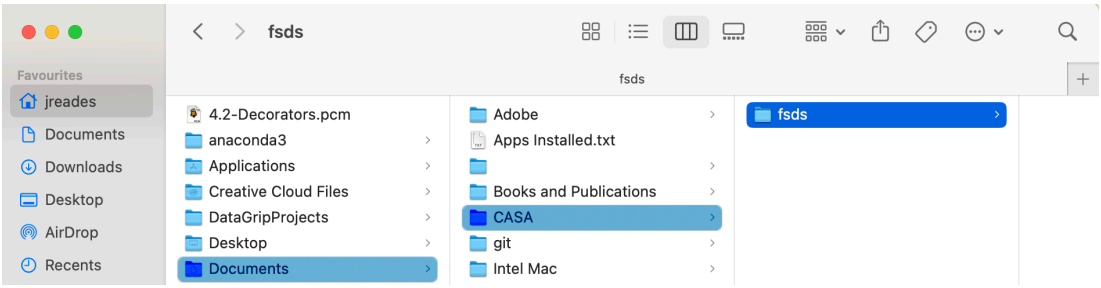
1.1 Starting Up 'Right'

One of the most confusing things about starting a Docker container with a "local volume mount" (i.e. a location on your computer that Docker *connects* to the container's file system) is that it seems like magic and it's often hard to understand why you're seeing what you are under the `work` directory.

So *before you do anything else* please spend a minute in the Terminal (macOS) or Power Shell (Windows) learning how to get to your home directory and, within that, to a CASA directory where you can store your work and keep Docker from accessing data that it shouldn't.

What we are doing is creating a directory on your computer that you can access from Docker. This is where you will store your notebooks, data, and any other files that you need to work with. The ultimate structure we'll produce this:

Figure 1: 'Target' Directory Structure



On both a Mac and a PC you should be able to run the following:

1. `cd $HOME` – this will **c**hange **d**irectory to your *home* directory (on a Mac it will be `/Users/<your username>`, on a PC it will be something like `C:\Users\<your username>`). *Hint:* `cd` *means 'change directory'!*
2. `cd Documents` – this will move you into your 'Documents' folder. *Note: on Windows this* ***might*** *be called* `My\ Documents`*, in which case it's* `cd My\ Documents`*! If you have set up your computer in another language this might be called something else, but Power Shell still 'knows' which folder should contain your documents.*
3. `mkdir CASA` – this will create a `CASA` folder in your home directory.
4. `cd CASA` – you are now changing into the `CASA` directory.
5. `echo $pwd` (PC) or `pwd` (Mac) – this should show you the 'full path' to your new `CASA` directory (e.g. `/Users/<your username>/Documents/CASA` or something like that).

Leave the Terminal or PowerShell window open! You will need it in a moment.

Perhaps a video will help clarify?

https://www.youtube.com/embed/5IkwUrYTY78

1.2 Configuring the sds2024 Image

During the 'install festival' you should have installed Docker and, time permitting, 'pulled' the image appropriate to your system. If you haven't, then you should do so as a priority *now*.

1.3 Running Docker

By default, the best way to start Docker is from the Terminal or Power Shell.

1.3.1 On Windows

Using the *same* Power Shell copy and paste the following **all on one line**:

```
docker run --rm -d --name sds2024 -p 8888:8888
  -v "$(pwd):/home/jovyan/work" jreades/sds:2025-amd start.sh jupyter lab
  --LabApp.password='' --ServerApp.password='' --NotebookApp.token=''
```

> **Windows Commands and Docker**
>
> `$(pwd)` is actually a *command*, you are asking the Power Shell to use the current working directory* (`pwd` == `print working directory`) as the 'mount point' for the `work` directory. The Command Prompt doesn't support `pwd`, but the Power Shell *should*. You can check this by simply typing `pwd` and hitting enter (⏎) to see if you get an error.

1.3.2 On macOS

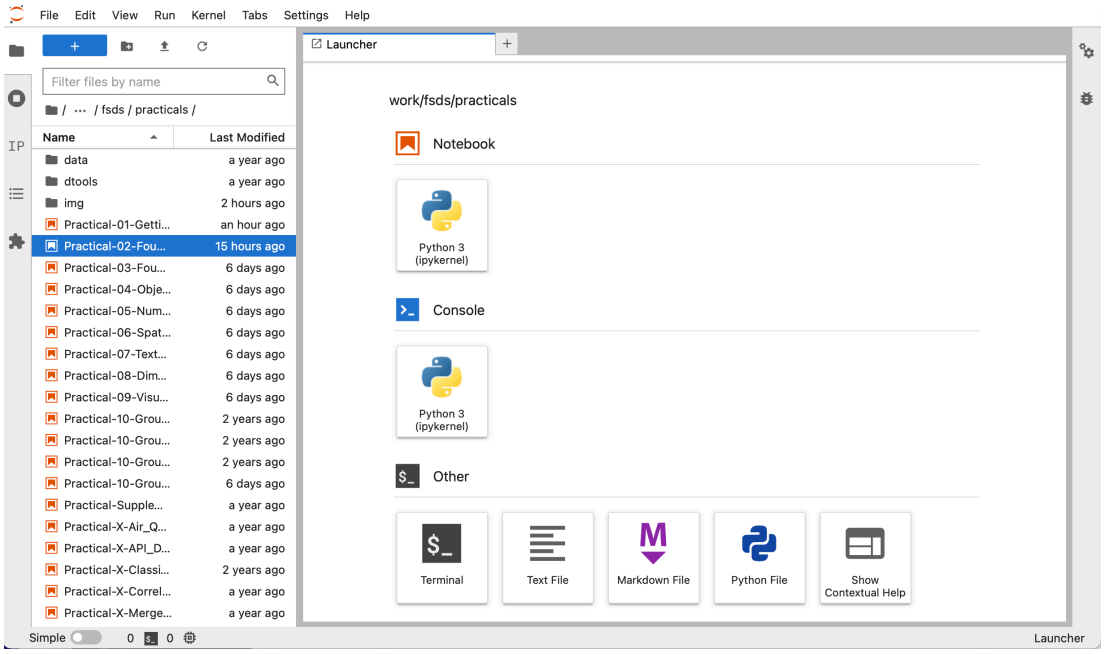
Using the *same* Terminal copy and past the following (change the docker image to `jreades/sds:2025-amd` if using an older Intel Mac):

```
docker run --rm -d --name sds2024 -p 8888:8888 \
   -v "$(pwd):/home/jovyan/work" \
  jreades/sds:2025-arm start.sh jupyter lab \
  --LabApp.password='' --ServerApp.password='' --NotebookApp.token=''
```

1.4 How do I Know it Worked?

With Docker running, you will *mainly* interact with Python through a web page. To check if it's running, we just have to visit the web page and see what happens: http://localhost:8888/lab/tree/work/. We'll talk more about exactly what is going on over the next few weeks, but this should show you a page that looks something like this (probably with fewer files listed on the left-hand side):

Figure 2: Screenshot of Jupyter Lab



> **See Docker Run (and Run)...**
>
> Once you have started a Docker container using any of the above, the machine **will continue running** until you either restart the computer or tell Docker to stop the container. This *can* consume memory and battery power indefinitely.

2 Setting Up GitHub

Understanding how to use Git and GitHub effectively is a core element of learning to code. So one of the *first* things that we are going to do is set you up with an account and a new repository.

So in order to complete this task you need to do the following (**more detailed explanations below**):

1. Create a login with GitHub (if you've not done so already).
2. Create a new **private** repository on GitHub.
3. Edit the `README.md` and `.gitignore` files for your new repository.
4. Save the changes (this is called a 'commit') and explain in a general way what edits you did.
5. Work out how to compare the original and edited versions of any file in your browser.

2.1 Your New GitHub Account

It doesn't really matter which way you do this, but we recommend that you set up your new GitHub account with *both* your personal and your UCL email addresses. GitHub 'knows' about educational users and will give you access to more features

for free if you associate a `.ac.uk` email address to your account. So choose one email address to start with and then add the other one later.

From a security standpoint you should *also* enable 2-factor authentication so that you receive a text message when you log in on a new machine and are asked to confirm a code.

2.2 Creating a Private Repository

To create a repository, click on the `+` at the upper-right corner of the GitHub web page and select `New Repository`. You might as well call your 'repo' `fsds` or `foundations` since that's a lot shorter than `foundations_of_spatial_data_science`.

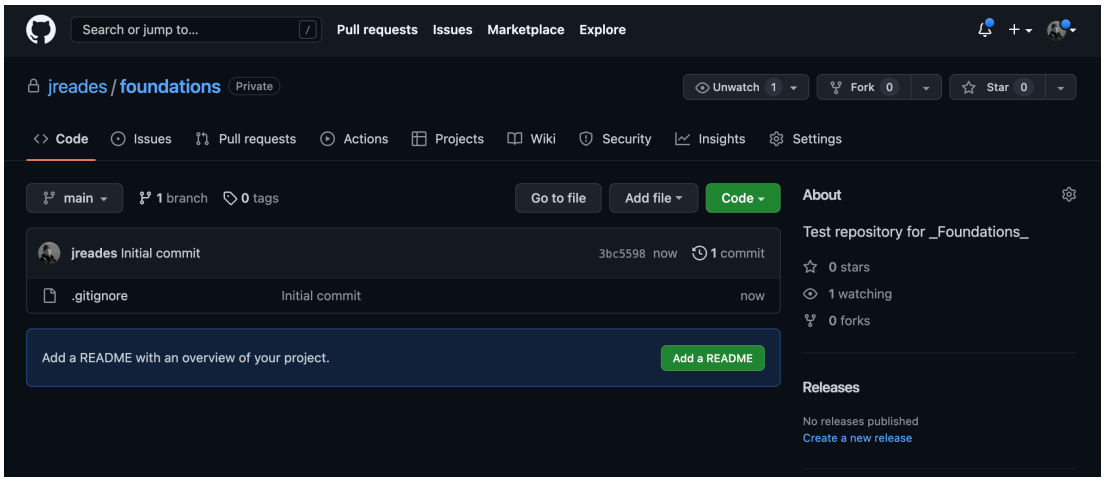
> **!** Your 'Repo' Name
>
> For the purposes of this tutorial (and all subsequent tutorials) I will assume that your repository is called `fsds`. You can call it whatever you like, in which case you will *always* need to substitute the name that *you* chose wherever you see me write `fsds`.

It's always helpful to provide some basic information about what's in the project (*e.g.* your notes and practicals for the *Foundations* module). And finally, make sure you:

1. Change the visibility from `Public` to `Private`,
2. Tick `Add a README file`,
3. Change `Add .gitignore` from `None` to `template: Python`.

Click `Create Repository` and you should end up on a page that looks like this:

Figure 3: Repository created



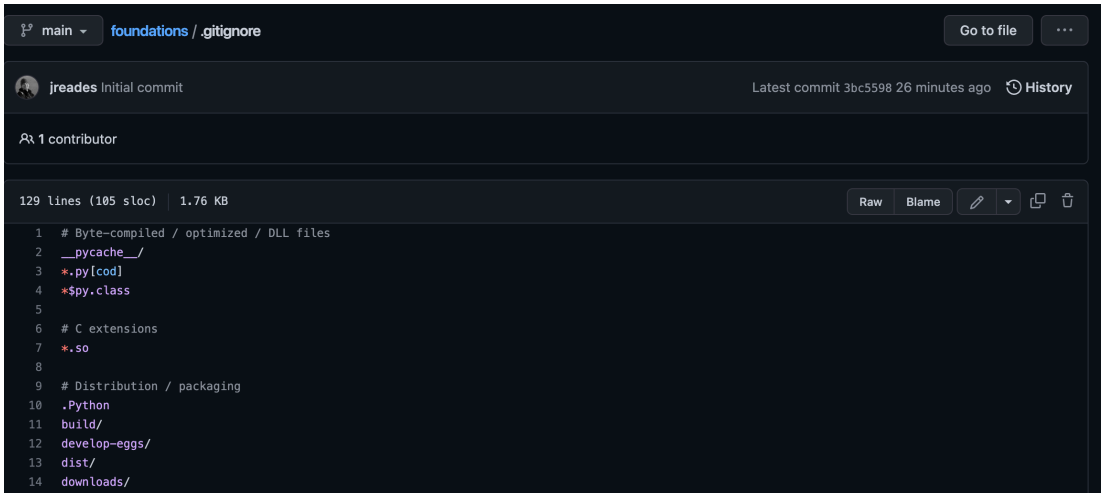
Your new repository has been created (on GitHub)!

3 Updating the .gitignore File

The `.gitignore` file tells Git what files to ignore by default. Unless you *force* Git to add an ignored file it will happily live in your *local* repository alongside files that are version-controlled and 'shared' with GitHub.

The Python template for `.gitignore` includes a lot of useful files and folders that we wouldn't want Git to track for us. But it *doesn't* include data. In your web browser, click on the `.gitignore` file and then the 'pencil' icon on the right to edit it on GitHub. You should see something like this:

Figure 4: Editing the .gitignore file



3.1 Exclude Data Files

We want to make it hard to accidentally add a large data file to our repository. Git/GitHub isn't designed for large, binary files (you can't 'read' a Parquet file) and we assume that data is backed up or available elsewhere, but our *code* is not! So as a first step we want to exclude files that are likely to just be 'data':

File Type	Extension
CSV	`.csv`
Excel	`.xls`, `.xlsx`
Zip	`.zip`
GZip	`.gzip`
Feather	`.feather`, `.geofeather`
Parquet	`.parquet`, `.geoparquet`

Here's another hint: `C extensions` are *already* excluded by the `.gitignore` file, so maybe look to see how that's done to help you figure out how to exlcude `.zip`, `.gz`, and `.csv` files...

3.2 Exclude a Data Directory

To make it even *less* like that we accidentally include data, let's also exclude a `data` directory from our repository. As a clue, nearly everything in the `Distribution / packaging` section of the `.gitignore` file is a directory to be excluded from Git.

So how would you indicate that `data` is a directory? Once you're sure, add the `data` directory!

When you are done, don't forget to add a 'commit message' (e.g. 'Added data files to .gitignore') at the bottom and then click `Commit changes`.

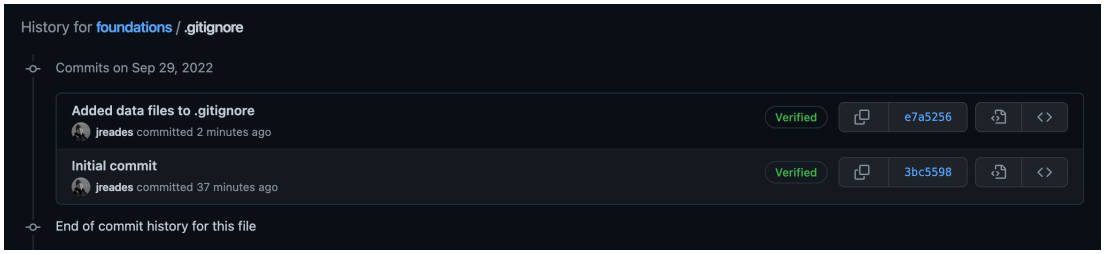
> **Quick Answer**
>
> I don't want you to get hung up on this *one* thing in Practical 1, so if you just can't make sense of what you're being asked to do here, have a look at the Answers at the bottom of this page.

3.3 Check Your Changes

Once you have committed your changes, you should be back to the default view of the `.gitignore` file but there should be a message to the effect of `Latest commit <some hexadecimal number> 10 seconds ago` and, next to that, a `History` button.

Click on 'History' and let's go back in time!

Figure 5: The Gitignore history



On the history page you can browse every edit to your file. Whenever you `commit` a file, this like taking a snapshot at a point in time. Using the 'History' you can compare two different snapshots in order to see what has changed. This would help you to work out how you broke something, check that requested changes have been made, or see how an error might have been introduced.

> **Viewing Your Commit History**
>
> You can mouseover the buttons to see what they do. Why don't you try to find `See commmit details` and check what edits you made to the `.gitignore` file? You should see at least *three* plusses in the history view representing three new lines in the `.gitignore` file.

4 Creating Your First Remote File

To get some practice with Markdown let's write up some notes direclty into our GitHub repository (aka 'repo'). You'll notice that we've not yet hit the *BIG GREEN BUTTON* marked `Add a README`... Let's do that now!

This will take you to an editing page for the new `README.md` file. You can type directly into this web page and it will update the repository, but *only* once you `commit` your edits.

4.1 Working on Your Markdown

Write your `README` file using *at least* the following Markdown features:

- A level-1 header (`#`)
- A level-3 header (`###`)
- Italic text (`_this is italicised_`)
- Bold text (`**this is bold**`)
- A link (`[link text](url)`)
- An image (`![Alt text](image_location)`)

If you're unsure how these work, just double-click on *this text* and you'll see Markdown in a Jupyter notebook. Here's some sample text to get you started:

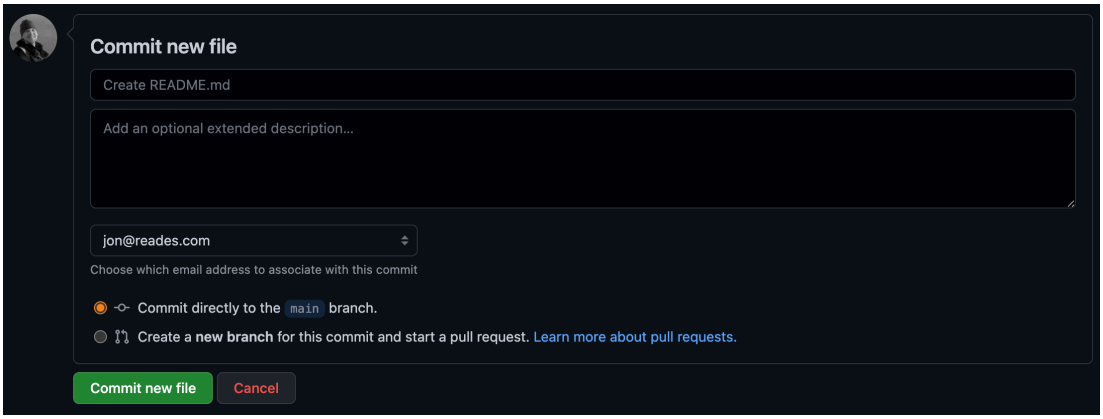
```
### Foundations of Spatial Data Science

This repository contains practicals and notes from the _Foundations_ module.

You can find the original [here](https://jreades.github.io/fsds/).
```

Don't forget to check out the "Preview" tab!

4.2 Commiting a Change

Once you're happy with how your text looks and works, it's time to commit! Scroll down to where you see something like this (you will see your *own* GitHub username, not mine):

Figure 6: GitHub Commit



You can just accept the description (*e.g.* `Create README.md`) or you can write your own. You can also provide an extended description if you choose. Then click `Commit new file` and you will see your new README appear.

5 Setting Up Git Locally

I have created a video on Microsoft Streams that runs you through the tasks below. You can refer to it if you find the written instructions hard to follow for any reason or would like a visual check that you're doing the right thing!

5.1 Configuring Defaults

The first thing to do is set up the default username and email for GitHub. These can be changed on a project-by-project basis, but to begin with it's best to set up the *global defaults*. Using either the Terminal or Bash enter the following (replacing `<...>` with *your* details):

```
git config --global user.email '<your GitHub email address>'
git config --global user.name '<your GibHub username>'
```

> **Recall: Convention!**
>
> As a reminder, `<...>` is a convention in programming to indicate that you should replace the text between the `<` and `>` with something that makes sense in your context. For example, if you see `'<your GitHub email address>'` you should type, for *example*, `'j.reades@ucl.ac.uk'`. You do *not* include the `<` or `>` characters!

5.2 Creating a Personal Access Token

For copying changes up to/down from GitHub, you *must* use a Personal Access Token. This is like issuing special passwords that allow only limited access to parts of your GitHub account.

To create a Personal Access Token:

- Visit *your* GitHub User Page (e.g. github.com/jreades)
- Click on *your* user icon (at the top-right corner of the page) and click the `Settings` link (near the bottom on the right side of the page).
- Scroll down the settings page until you get to `Developer settings` (near the bottom on the *left* side of the page).
- Click the `Developer settings` link to reach the 'apps' page and then click on the `Personal access tokens` link.
- Select `Tokens (classic)` from the left-hand menu and then click the `Generate new token` button.

> **Types of Personal Tokens**
>
> You now need to choose the type of token to generate. I *personally* find the old type of tokens easier to work with because the 'new' fine-grained tokens are intended to support complex workflows when all we're trying to do is allow one computer to push/pull from Git.

- `Generate new token (classic)` for general use token and then specify the following:
  - I'd suggest writing `FSDS Token` or something similar in the `Note` section.
  - Set the expiration to `90 days`
  - Click the `repo` tickbox for 'Full control of private repositories'.
- Save the resulting token somewhere safe as you will need it again! (e.g. a Note on your phone, a password manager, etc.).

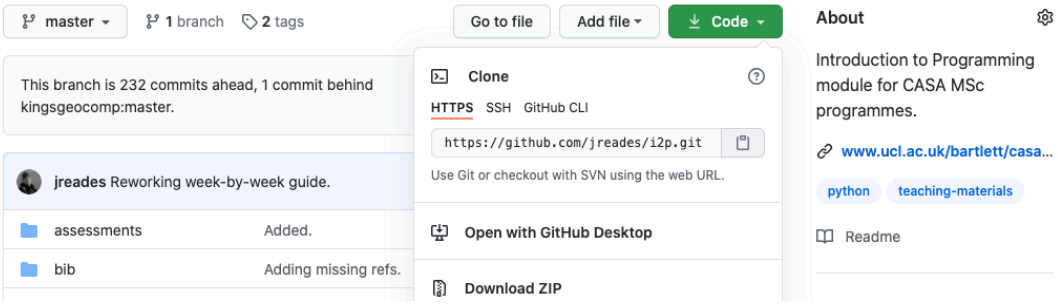
> **Keep your Personal Token Safe**
>
> You will need it at least twice in this tutorial and may want to use it again on other computers. You can always create a new one, but then you'll need to update every computer where you access your GitHub repositories.

5.3 Cloning Your Repository

Now we are going to `clone` (i.e. copy) the repository that you just created on to your own computer. This is surprisingly straightforward provided that you have installed the command line tools.

On your private repository page, click on the green button labeled `Code` visible in the screenshot below:

Figure 7: Screenshot of cloning interface



You should then copy your HTTPS URL (in my screenshot it's `https://github.com/jreades/i2p.git`).

Now go back to the Terminal or PowerShell that you left open earlier and type the following (replacing `<the_url_that_you_copied_from_the_browser>` with the URL that you copied from GitHub):

```
git clone <the_url_that_you_copied_from_the_browser>
```

The first time that you do this, you will need to provide login information. Use your GitHub username and the Personal Access Token that you just created. On your computer you should now see a new directory with the same name as your repository. For example: `Documents/CASA/fsds`.

5.4 Storing Credentials & 'Pulling'

You can now activate the credtial helper that will store your Personal Access Token (though you should *still* keep your secure note!):

```
cd fsds
git config credential.helper store
git pull
```

When you type `git pull` you *should* be asked *again* for your username and password. You should (again) use the Personal Access Token as your password. You should not be asked again for pushing or pulling data into this GitHub repository. If you are *not* asked for your Personal Access Token then this likely means that your token is already saved and ready to use on all future 'actions'.

5.5 Jupyter < - > Your Computer

If you now go back to Jupyter, you should now see that the `work` directory is actually the same as the `CASA` directory that you created earlier; you know this because you can now see an `fsds` directory containing that wasn't there before.

This is because Docker is 'mounting' the `work` directory in the container to the `CASA` directory on your computer. This means that you can save files in the `work` directory in Jupyter and they will be saved in the `CASA` directory on your computer.

> **Organising Your Work**
>
> There are any number of ways to organise your CASA work, what's important is that you are *logical* about things like names and hierarchy. This will make it much easier to access files and notebooks using Docker, Quarto, *and* Python.

5.6 Adding a Local File to Your Repository

In order to tie together the different concepts covered above, we are now going add Practical 1 (*this* practical) to *your* GitHub repo. The easiest way to do this is to download the notebook by clicking on the `Jupyter` link on the right side of this page. So the process is:

1. Click on the Jupyter link to save this file to your computer as a notebook with the extension `.ipynb`.
2. Move the file to your new repository folder (*e.g.* `$HOME/work/Documents/CASA/fsds/`).

> **File Extensions**
>
> It is *highly* likely that your browser automatically added a `.txt` extension when you saved the Notebook file to your computer. You need to remove that ending to your file name or Jupyter won't be able to run it. You can rename a file by either doing it directly in the Finder/Windows Explorer, or by *moving* (bash: `mv`) the file: `mv <notebook_name>.ipynb.txt <notebook_name>.ipynb`

In the Terminal/PowerShell we now need `add` this file to Git so that it knows to keep track of it. Unlike Dropbox or OneDrive, just putting a file in a repo directory does *not* mean that Git will pay attention to it:

```
# Assuming that you are 'in' the 'fsds' directory...
git add Practical-01-Getting_Started.ipynb
git commit -m "Adding notebook 1 to repo."
```

> **Add, Commit, Push, Repeat**
>
> Unless you have *added* and *committed* a file to Git then it is *not* version controlled.
> Unless you have *pushed* your committed files to GitHub they are only backed up locally.

5.7 Status Check

We now want to check that the file has been successfully added to Git. We do this with a `status` check in the repository directory (*i.e.* `cd $HOME/work/Documents/CASA/fsds/`):

```
git status
```

You should see something like:

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
```

This is telling you that your local computer is 1 commit (the one that you *just* completed) ahead of the 'origin', which is on GitHub. GitHub doesn't *have* to be the origin (nor does the repository have to be one that we created in order to be an origin) but *conceptually* and *practically* it's easier to create new repositories on GitHub and clone them to our computer.

5.8 Keep Pushing

To synchronise the changes we just made, let's follow Git's advice:

```
git push
```

You should see *something* like the below series of messages (the exact details will differ, but the 'Enumerating, Counting, etc' messages will be the same):

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 306 bytes | 306.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote: This repository moved. Please use the new location:
remote:   https://github.com/jreades/fsds.git
To https://github.com/jreades/fsds.git
   7410d0e..45aa80a  master -> master
```

If you now go over to your browser and visit your GitHub repo page (*e.g.* `https://github.com/jreades/fsds`) — pressing the `Reload` button if you had the page open already — then you should see that the file you added *on your computer* is also showing up on the GitHub site as well! This means it's now fully version-controlled and backed-up.

> **Keep Pushing**
>
> Unless have *pushed* your commits to GitHub they are *only* stored on *your* computer. So your files can be properly version-controlled, but without a *push* if you lose your computer you *still* lose everything!

5.9 More About Git

From here on out you can keep changes made either directly on GitHub or locally on your computer (or any *other* computer to which you `clone` your repository) in synch by using `git push` (to *push* changes from a local computer *up* to the origin on GitHub) and `git pull` (to *pull* changes available on the origin *down* to the local computer).

That said, you can do a *lot* more than just push/pull to your own repository and this Twitter thread leads to a lot of useful additional resources to do with Git:

- Introduction to Version Control with Git on the Programming Historian web site is written for digital humanities researchers so it's intended to be accessible.
- Oh My Git is an 'open source game' to help you learn Git.
- Git Meets Minesweeper? is apparently a 'thing'.
- Visual Git Reference if you think visually or just want to check your understanding.
- Version Control with Git is a Software Carpentries lesson that takes you quickly through the important elements of getting set up and started. It would be a good refresher.
- Altassian's Documentation provides more detailed explanations of the commands and options.
- Learn Git Branching focusses on a key concept for software *collaboration*.
- Git Immersion provides a 'guided tour' of the fundamentals.

> **Tip**
>
> For the Group Work every member of your group will need to make contributions to a GitHub repository. This will require learning how to invite others to be contributors, how to merge changes, and how to deal with conflicts of the coding kind.

6 Setting up a GitHub Web Site

> **Bonus Content**
>
> It is not core to this module, but if you'd like to really make the most of Git, why not create a GitHub.io web site? Using Markdown (for the most bare bones effect) or code (via Quarto), you can render and push a web site to `<your GitHub username>.github.io/`. You might find this useful for creating a simple portfolio, blog, or other collection of 'outputs' that you can use to demonstrate to employers that you know your way around.

Rather than create a GitHub.io site using your *Foundations* repository, we'd recommend building one in the separate 'main' repository so that, for example, jreades.github.io, returns a web page.

To do this, the steps are detailed here, but read on to get an overview:

1. On the GitHub web site, create a new repository called `<your username>.github.io` (where `<your username>` is whatever your github username is). So for us, it would mean creating a *new* repository called `jreades.github.io`.
2. For your first web site, try setting up GitHub pages **without** specifying a branch: this way, *anything* that you put into your GitHub repository will show up on the public web site. Later, when you are more comfortable with GitHub, you may wish to switch to using a branch so that you can do work without *always* and *immediately* updating the web site.

3. One you've followed the GitHub instructions for publishing a web site, you should `clone` the site to your computer.
4. After cloning the site to your computer, try making a change to the `README.md` file and `push`ing the change back to GitHub. After a minute or two (possibly up to 10 minutes) you should see your web page update!

Congratulations, you can now publish a web site using nothing more than Markdown.

As you develop your Markdown, coding, and Quarto skills, you might also want to look into publishing to GitHub pages from Quarto. That's how we build the entire *Foundations* web site, so there's a *lot* further that you can go with this...

7 Other Ways of Using Docker

> Advanced Content
>
> The two techniques below are: 1) advanced; and 2) not essential to using Docker for this module. We will offer limited support for this if we can, but you will be a lower priority for support since these are *entirely* at your discretion.

There are two *other* ways of running Docker:

1. If you have a Mac or have installed a full Linux system for WSL2 (e.g. Ubuntu) on your Windows machine, then there is a 'bash script' and configuration file as detailed here that we created for you. This is the *easiest* way to start/stop Docker.
2. Using VSCode as detailed here to create a new VSCode project that is 'bound' to the Docker image. This will give you a very different experience of using Docker and Jupyter, but is appropriate for *those who want to use an IDE to write code*.

7.1 Other Useful Resources

- GitHub Markdown Guide
- Common Mark
- Markdown Guide, which helpfully includes do's and don'ts.

Finally, these are a bit overkill but the bits about setting up and installing `git`, `bash`/`zsh`, and so on may come in handy later:

- Setting Up a New Mac
- Beginner's Guide to Setting Up Windows 10
- Setting up Windows without Linux
- Microsoft Python Setup Guide

8 Answers

Normally, we will provide 'answers' later in the week, but for *this* week it makes sense to provide them right away if you need them...

8.1 .Gitignore

The main thing you should notice is the pattern: `*` means 'anything', while `/` at the end of a line implies a directory. So the following four lines should be added to your `.gitignore` file:

```
*.zip
*.gz
*.csv
*.gzip
*.feather
*.geofeather
*.parquet
*.geoparquet
data/
```

That's it.