Reproducible Work

What is reproducible work?

- Contains relevant code, including which packages were used
- Contains enough text, either via `markdown` or comments, to be able to understand what the purpose of each code is
    - Ideally integrates code and results, along with text, into a single document

Why is it important?

Simply put, mistakes happen. If your project if structured properly, you will have a code document that contains all of the relevant information, and it is easy to recreate the outcomes. That way, if you move computers, delete a document by accident, or hand over the project to another person, everything is neatly contained and can be reproduced without hassle.

How do I make reproducible work?

There are many different ways to make reproducible work. The information listed here should give you the foundations upon which you can build your own systems. However, the principles are the same, and largely revolve around project structures and a version control system, such as `git`.

Structuring a project

This is the structure that I find works for me. You may want to find a variation on it that works for you, but the basic premise of keeping repositories self-contained should remain.

```
proj/
├── data/
├── docs/
├── figs/
├── funs/
├── out/
├── cleaning.R
└── analysis.R
```

As you can see, the project repository contains separate directories that you can use to store different file types. Importantly, the analysis and cleaning files are stored in the project root, allowing easy use of relative paths over explicit paths e.g. `read_csv(here('data', 'data_file.csv'))` rather than `read_csv('C:/Users/owner/Documents/Repos/my_project/data/data_file.csv')`. The reason why we prefer relative paths is that they allow projects to be used by multiple people without the need to re-write code. If you change computer, or the project is opened by another person, the code will break as they will not have the same directory structure as the computer that the code was created on.

> **Note:** the example above used an `R` package called `here_here`, calling the function `here()`. Similar solutions may exist for other languages, and you should try and find them for the language of your choice.

`data/`

An important idea is that you should treat your data as read-only. You and your team have likely worked hard to collect the data and it's easy to make a changes along the way that you either forget about, or need to reverse. As most projects span a long time between the data collection and analysis stages, if that happens to you it will take a lot of work to figure out exactly which changes you are interested in reversing etc. To save yourself this hassle, and help make your work reproducible, once the data is collected it should not be edited; all the work should happen in your code, allowing it to be easily checked.

Other subdirectories

- `docs/`: this contains the output documents. For example, if you are using

`R Markdown` to create a pdf via `LaTeX`, you could place them here.
- `figs/`: this contains the functions you write and might want to reference. The idea is to create functions so that can give code a meaningful name. It also helps if you need to repeat a code chunk multiple times, especially if you need to edit it at some point, as you can just call the function rather than typing it out each time.
- `out/`: this contains files that are produced from the original data e.g. cleaned data files. You can then call them in your analysis scripts.
- `figs/`: this contains figures that may be generated from your scripts.

Importantly, if you follow the principle that your `data/` files are read-only, all of the files in these directories (with the exception of `funs/`) *should* be reproducible and could be deleted at any time without concern of generating them again. In order to revert to previous figures and output versions, you will need to be able to track changes in your code. This is where a *version control system* like `git`, which we will discuss in the next section.

File names

How you name files and directories may not seem like an important point, but it can cause quite a headache if you try and use code to automate processes, and at best, it just slows things down. To quote Aaron Quinlan, a bioinformatician, "a space in a filename is a space in one's soul".

Instead try and use something like this.

- KISS (*Keep It Simple Stupid*): use simple and consistent file names
    - It needs to be machine readable
    - It needs to be human readable
    - It needs to order well in a directory
- No special characters and **no spaces**!
- Use YYYY-MM-DD date format
- Use `-` to delimit words and `_` to delimit sections
    - i.e. `2019-01-19_my-data.csv`
- Left-pad numbers
    - i.e. `01_my-data.csv` vs `1_my-data.csv`
    - If you don't, file orders get messed up when you get to double-digits

Key Points

- Use a version control system such as `git` to track changes in your code.
- Data isn't touched one collected:
    - Do all *data munging* within your program i.e. no editing the excel spreadsheets!!!
- Never set explicit file paths (e.g. `setwd()`) if you can avoid it
    - Try and use a package that allows you to set relative paths e.g. `here_here` in `R`. This allows the project to be passed to someone else in its entirity and the code won't break because they don't have the same folder names and set up as you (also if you work on multiple computers/OS)
- Format your filenames properly

Git

Think of it as tracked changes for your code. When working on a project by yourself, it's important to be able to go back to previous versions if you make a mistake and can't remember all the steps you went through since your last stable version.

Set up

There are many ways to get `git` running on your computer.

- SourceTree
- Remotes vs local
- Basic commands
    - commit
    - push/pull
    - fetch

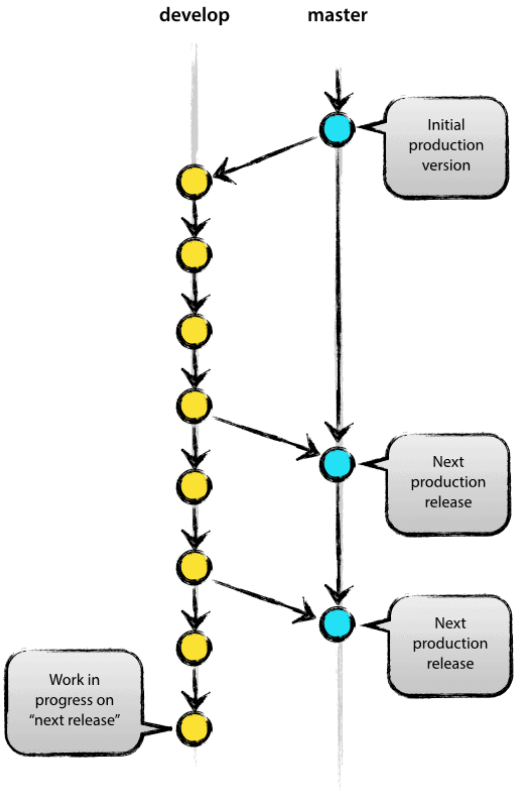
Branching

Branching is a key part of the `git` workflow. It allows you to make changes to your code, without worrying about breaking previously 'good' code. But what is it?

Simply put, when you create another branch you are creating a copy of your code at that point in time. This is useful because it allows you to make changes to your copy, and leave your original code intact! So there's no concern about breaking your working code while you test out some ideas.

> "But isn't that why we use git?"

Kind of. But git is only so powerful. If you have working code, you don't want to put it out of action whilst you test ideas out, especially if other people need to use your code and can't wait for you to figure out your future problems. So creating another branch allows you to get around this issue. For most projects, you can get away with just two branches, a `master` and a `develop`, which are explained below. If your project is complex, and requires multiple people to work on the code at the same time, it would be worth you looking at implementing this model.



The image to the right is copied from the model listed above. It is useful in illustrating the `master` and `develop` approach to branching.

Creating a branch

As with all things in `git`, you can do this multiple ways. I prefer to use the SourceTree application, as I find it far more intuitive when you can see the changes, but you can use the command line or git bash. If you want to explore the command line code, I would recommend visiting this website, which allows you to interact with the code through illustrations.

Notebooks

- Many examples of different kinds of notebooks
- Jupyter notebooks are useful as you can see the results immediately integrated within the document
    - Have a system to allow other statistical and programming languages to run via kernels
        - Other systems don't allow this level of integration, so will not be explored in much detail
- Jupyter labs is the updated version of notebooks that acts as a full IDE (integrated development environment), allowing you to

Jupyter notebooks

This section will give you a brief overview of what a Jupyter notebook is and how to use them, but if you would like a more detailed understanding, please read the official documentation.

Jupyter notebooks are run on `python`, though additional things can be downloaded to allow you to use your programming language of choice. For an example of what you can do with Jupyter notebooks, click here, and here for a collection of neat and applied notebooks.

Mac's come shipped with a version of `python`, but it is most likely outdated, and it doesn't contain everythin we want. In order to get running, I strongly recommend downloading the `anaconda` distribution over other distributions, or even just directly from `python's` website. The instructions below will be enough to get you up and running with Jupyter notebooks in your language of choice.

- Download the full anaconda distribution i.e. not miniconda
    - Be sure to choose `Python 3.x`, not `Python 2.x`, as it's the newer version and is fowards-compatible.
    - Be sure to only install for one user, not the whole system
    - Be sure to select `Add Anaconda to my PATH environment variable` under Advanced Options
    - Be sure to install `Anaconda` to the `H:\` drive on your computer, as this is where your data lives. To do this you will need to manually edit the

installation path within the `anaconda` installer wizard, otherwise it will end up in the `C:\` drive
        - This is OK if you are able to store data on this drive, and therefore can create your repositories within the `C:\` drive.
        - Worst case scenario you can use the command `cd "H:/..."` at the top of the notebook to specify the relevant path to your data, but this is bad practise for the reasons mentioned [above](# Structuring a project).
- Use `kernels` to connect your programming language of choice with python and the notebook
    - To see how to get a particular language to work in Jupyter Notebooks, please click on the appropriate language:
        - Stata
        - SAS
        - R
        - Other kernels
- Visualization for data exploration:
    - plotly (https://towardsdatascience.com/the-next-level-of-data-visualization-in-python-dd6e99039d5e)

Creating a notebook

> **Note:** You can substitute the phrase *"Jupyter notebooks"* with *"Jupyter Labs"* if you would prefer to have a full IDE allowing you more control over the system.

You can either open up the anaconda navigator and then Jupyter notebooks, or open Jupyter notebooks directly. Once open, navigate to the directory you would like to create the notebook in (*If you are using a version control system like git, then you should be within the project's repository*)

Select the **New** button in the top right corner, and then select the language you would like to program in (*this assumes that you have downloaded an appropriate `kernel` if you would like to use a language other than `python`*)

Installing the Stata Kernel

The instructions for installing the `stata_kernel` are based from the original documentation here. It should work with `Stata 12` (we have tested it). If these instructions do not work for you, it may be that there has been an update to the `kernel`, at which point, please refer to the original documentation linked above.

Open a command prompt (Windows) / terminal (linux/mac) and type/copy-paste the following commands, pressing enter after each line

- `pip install stata_kernel`
- `python -m stata_kernel.install`

Windows-specific steps

In order to let `stata_kernel` talk to Stata, you need to link the Stata Automation library:

1. In the installation directory (most likely `C:\Program Files (x86)\Stata12` or similar), right-click on the Stata executable, for example, `StataSE.exe` (this will just show as `StataSE`, but is listed as an application). Choose `Create Shortcut`. Placing it on the Desktop is fine.
2. Right-click on the newly created `Shortcut to StataSE.exe`, choose `Properties`, and append `/Register` to the end of the Target field. So if the target is currently `"C:\Program Files\Stata12\StataSE.exe"`, change it to `"C:\Program Files\Stata12\StataSE.exe" /Register` (note the space before `/`). Click OK.
3. Right-click on the updated `Shortcut to StataSE.exe`; choose Run as administrator.
4. Enter your CIHS details

Installing the SAS kernel

This has not yet been tested here at PHO. The instructions for installing the `sas_kernel` are based from the original documentation here

Open a command prompt (Windows) / terminal (linux/mac) and type/copy-paste the following commands, pressing enter after each line. First we need to install a dependecy called `saspy` that helps the kernel connect `SAS` to `python`

- `pip install saspy`
- `pip install sas_kernel`

You should now see something like this.

```
Available kernels:
python3    /home/sas/anaconda3/lib/python3.5/site-packages/ipykernel/resources
sas        /home/sas/.local/share/jupyter/kernels/sas
```

Now verify that the SAS Executable is correct

- find the `sascfg.py` file -- it is currently located in the install location (see above) `[install location]/site-packages/saspy/sascfg.py`. To query `pip` for the location of the file, type `pip show saspy`. Failing that, this command will search the OS for the file location: `find / -name sascfg.py`
- edit the file with the correct path the SAS executable and include any options you wish it include in the SAS invocation. See examples in this file

Connecting R with Jupyter

If you are hoping to make nice documents and reproducible work using `R`, I would highly recommend that you use the `R Markdown` or `R Notebook` through `RStudio` application instead. However, if you would prefer Jupyter, then please read on.

It is possible to download an `R kernel`, much like for `Stata` and `SAS`, but it can be a bit fickle, so a different approach is described below. It is important to note that with this method you are installing a fresh version of `R`, so you will not have access to the packages you have previously installed - you will need to reinstall them in this `R` environment, which could be done within a Jupyter notebook.

Open a command prompt (Windows) / terminal (Linux/Mac) and type/copy-paste the following commands, pressing enter after each line

- `conda install r-essentials r-igraph`
- `Rscript -e 'install.packages("languageserver")'`

If you would rather install an `R kernel` than a fresh install of `R` within the `anaconda` distribution, you can follow the instructions here. The advantage of this is that it allows the

notebook to access previously installed packages as they are not running off a fresh version of `R`.

Connecting other kernels

To see a full list of `kernels` available for Jupter, along with the appropriate documentation and installation instructions, follow this link.

Tidy data

What do I mean by tidy data?

- tidyverse info

Additional resources

This document only touches on enough to get you up an running with reproducible work. However, to become fully proficient you will need to delve deeper into the material - trust me, it'll make your life easier in the long run. Here are a few places to start for each section, many of which were the basis for the systems I implement and advocate for here.

Project structure

https://nicercode.github.io/blog/2013-04-05-projects/

https://tomwallis.info/2014/01/16/setting-up-a-project-directory/

> These two resources describe the project structure that I advocate for in a little more detail, with a few minor differences. If you work in `R` I would recommend that you follow nicercode's other suggestions as well, particularly regarding the creation of an R project.

https://medium.freecodecamp.org/why-you-need-python-environments-and-how-to-manage-them-with-conda-85f155f4353c

> This takes a deeper look into how to manage `python` environments with anaconda, and

> how this affects your project structures. This is useful if you are work with `python 2.x` and `python 3.x`, but also allows your to ensure old code won't get broken when modules are updated as each module is specific to the environment it is downloaded in.

Git

https://happygitwithr.com/

> I cannot emphasise this enough: this is genuinely **the best resource** I have come across for explaining how to set up and organise a project with `git`. Whilst it is aimed at `R` users, there is a large amount of cross-over, so read it regardless of the language you use.

https://medium.freecodecamp.org/how-not-to-be-afraid-of-git-anymore-fe1da7415286

> This helps you understand the nuts-and-bolts of `git` by learning to use the command line, rather than an application like SourceTree.

https://git-scm.com/book/en/v2/

> The literal book on git. Everything from the basics to the advanced.

https://nvie.com/posts/a-successful-git-branching-model/

> If you feel comfortable with the idea of branching and are interested in a good extension of what we've covered, this will help. Roughly speaking, the more complex you project is and the more people that are involved simultaneously, the more branches you want so you can handle problems as they come up, without breaking previously 'good' code.