Installing Jupyter Notebook

Jupyter Notebook is a fantastic tool that improves workflow and overall efficiency in research. Jupyter Notebook is a web-based application that enables the creation and editing of interactive notebook documents that contain live code, text, documentation, videos, equations, and references. The Notebook itself is created by Project Jupyter, it is open source and supports over 40 programming languages, including the traditionally used languages of social scientific research - Stata and R.

The fact that Jupyter Notebook can host live code, execute commands in any order, and present real-time results, alongside various documentation makes it an attractive tool for any researcher to use - especially when compared to the clunky often times inefficient nature of a .do file in Stata for instance.

This post could go on forever talking about how great Jupyter Notebook is, but let's leave that for another post. For now, let's focus on setting your machine up with Jupyter Notebook so you can explore it for yourself. As mentioned previously Jupyter supports over 40 programming languages - Python is one of them. Python is required to run Jupyter in its environment and as such needs to be downloaded also. Jupyter requires Python 3.3 or greater.

Jupyter Notebook and by extension, Python can be installed in one of two ways, either through Anaconda or through PIP.

Jupyter Notebook: Anaconda route

Head over to the Anaconda website and download the most up-to-date version. Go through the 'Getting Started' setup, select 'Just Me' when given the option to select installation type, select your installation location (note: make sure if you have multiple drives to select your 'main' drive), and also register Anaconda as your default Python in advanced installation options. After this simply wait for your installation to complete.

After your installation is finished, search for 'Anaconda Navigator' in your Start Menu. Click on the Install Jupyter Notebook button. After the installation is completed, click Launch Jupyter Notebook. At this stage, I would also advise you to right-click on the launched Jupyter Notebook icon and pin it to the taskbar (saves clicking multiple icons next time around).

Jupyter Notebook: PIP route (Preferred if on Mac OS)

(Note: You will need PIP at some point when installing packages into Jupyter, it is best to download it now)

PIP is a little more old-school. PIP is a package management software system used. to install and manage libraries of content that are written in Python. These files are stored in the Python Packaging Index (PyPI) or PIP.

First, we need to check if Python is installed on your machine by first searching for 'terminal' then hitting enter, then typing:

python --version

If Python is installed then you will see something like this:

Python 2.07.0

If it is not then follow these instructions if you have a Windows PC and these instructions if you have a mac OS.

Once this is installed you are ready to install PIP. Firstly, open the cmd terminal like before. Secondly, type the following:

pip3 install --upgrade pip

followed by:

pip3 install jupyter

This should install your system with the most up-to-date version of PIP. Following this, we now need to install Jupyter onto our machines. To do this we type the following command:

python -m pip install jupyter

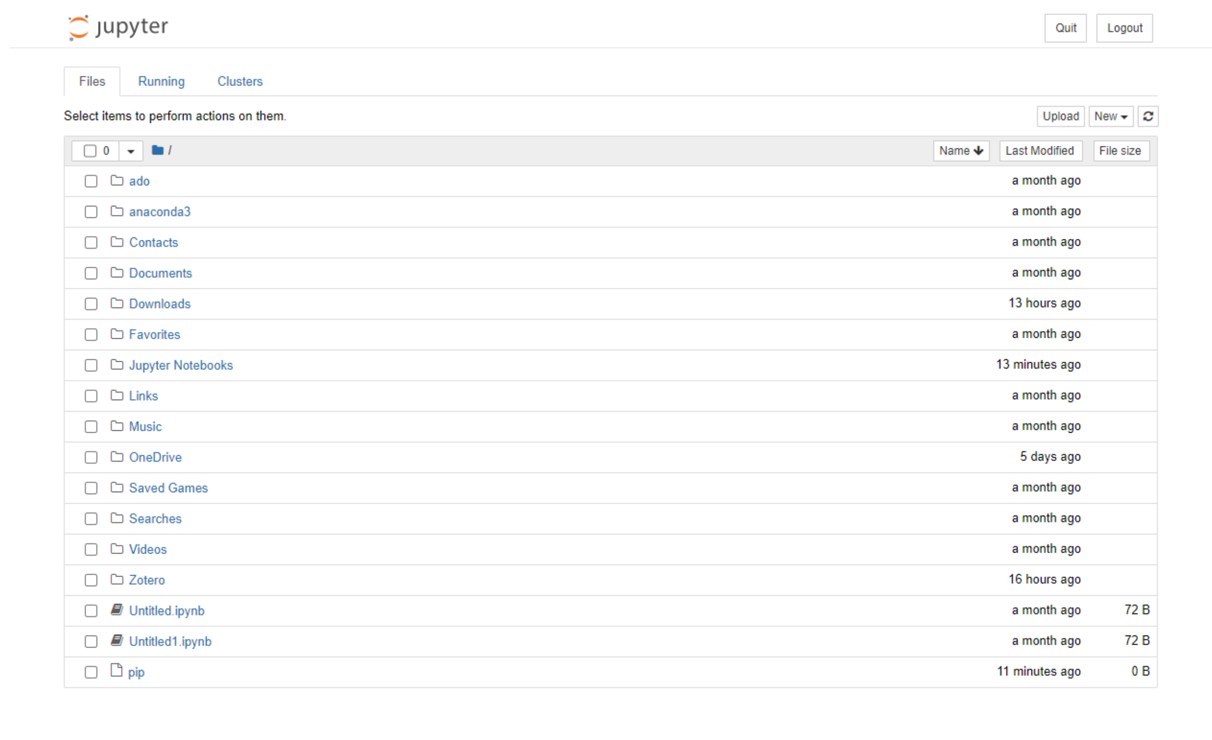
Following that, Jupyter Notebook should launch when you type:

jupyter notebook

Both these methods should work with both Windows and Mac OS devices. After you have finished downloading it is time to explore Jupyter Notebook!

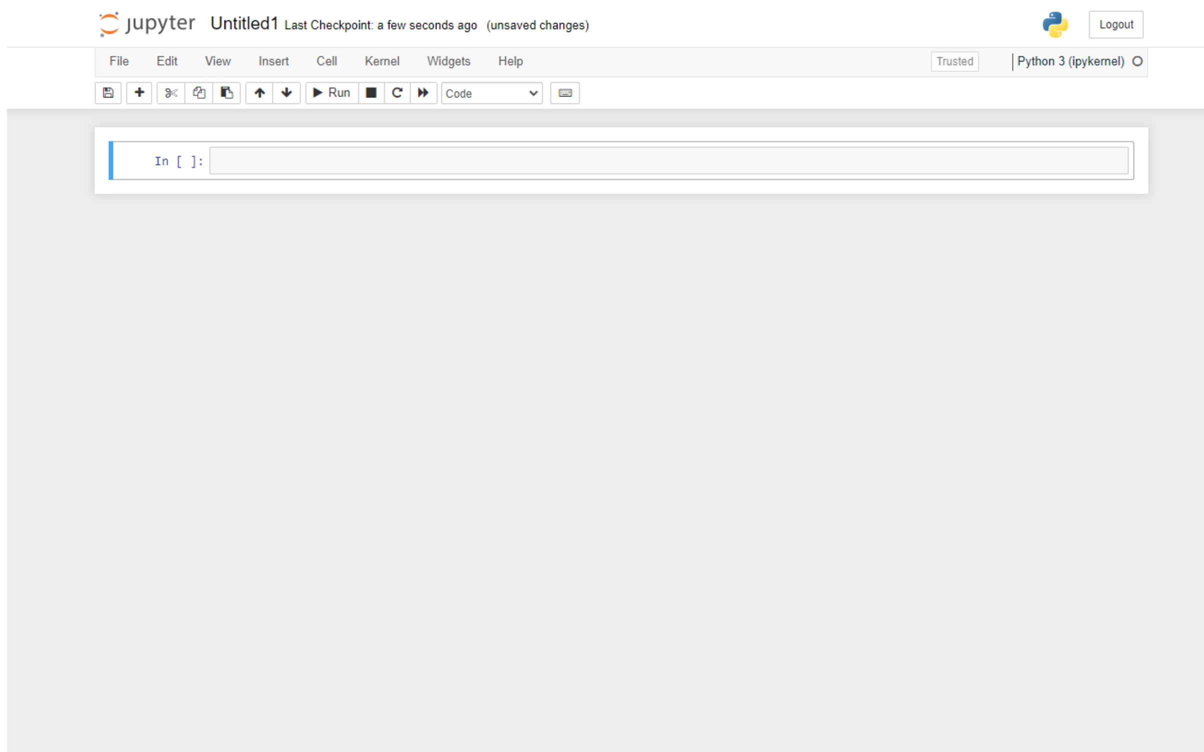
Getting setup in Jupyter Notebook

Now that we have successfully installed Jupyter Notebook, it is time to set up and explore the environment. When you first open Jupyter Notebook itself you will be presented with a screen not too dissimilar to this:



This can be considered your 'home page' for all things Jupyter. Jupyter should have opened a tab on your default browser, you may have also noticed that the URL for the 'homepage' is something that includes the words "localhost'' in it. Don't fret, localhost is not a website online, it simply indicates that the content is being served from your own machine on your computer - this is why we can use Jupyter Notebook for our research and not Jupyter Lab. In reality, it is the location on your local disk drive that you have selected to place Jupyter Notebook. This can be changed using the settings.

Now we are actually in Jupyter it is time to fire up a Notebook. Click new, then under kernel select 'Python 3 '(ipykernel)'. This will provide you with your very own Notebook:



The most obvious first impression of the Jupyter Notebook is that it looks like a cross between a Word Document and a .do file. Essentially that is what it is for the basics. If you head back to the homepage you will see that an instance of your Notebook has already been saved, typically something like 'Untitled.ipynb'. The '.ipynb' part is what the Notebook **is**. Each time a new Notebook is created, a new .ipynb file is also created.

Before we start showing off any tools etc we have one more piece of setup to complete: setting up Stata in Jupyter. For a guide on how to do this with R check this out using the Anaconda environment we set up earlier.

Stata Setup

For Stata 17 and above, this is relatively easy. Before typing anything into Jupyter, we first need to go back into our terminal and use PIP. In PIP we need to download something called the 'stata_setup':

```
pip install --upgrade --user stata_setup
```

This is the same in both Windows and MacOS enviroment.

Now we can go back to our Jupyter Notebook. We need to do two things using one code cell. The first is to:

```
import stata_setup
```

This, funnily enough, imports the setup procedure for Stata software. Secondly, we need to attach this setup to our pre-existing Stata software on our own devices. This is accomplished by writing:

```
stata_setup.config("directory_folder", "Stata_version")
```

To give you a working example of this, if my Stata18 program was stored locally in my C:/ drive and the copy of Stata I owned was the 'se' version - most commonly given out by universities, then I would type this:

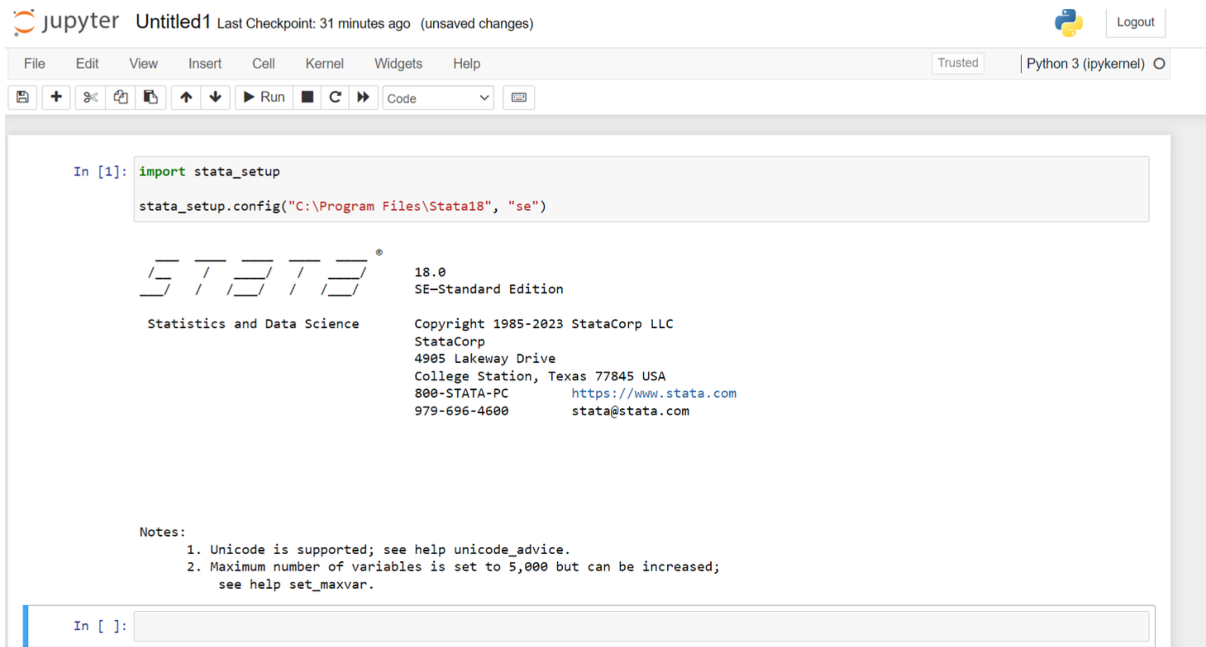
```
import stata_setup

stata_setup.config("C:\Program Files\Stata18", "se")
```

This should locate your Stata program and initiate setup. If you cannot locate the Stata directory yourself, open up Stata and type:

```
display c(sysdir_stata)
```

Copy the output into the "directory_folder" location.

If it does this successfully you will now see a live instance of Stata within your Jupyter Notebook environment:



This should provide you with everything you need to get underway!

Note on (very) old Mac OS:

If you have an older Mac OS that cannot be updated you may run into issues when installing Jupyter through both the Anaconda and PIP routes. Typically you will face an issue when launching through Anaconda - where it simply will not open, or with PIP via the terminal that provides a message "Segmentation Fault 11". This isn't an issue with Jupyter itself it is instead an issue the pyobjc-core package. To fix this, type in the terminal:

pip install pyobjc-core pyobjc-framework-cocoa --force-reinstall

This should then fix any issues you have and you can then open Jupyter Notebook via:

jupyter notebook

Exploring Jupyter Notebook

Now that we have gone through the tasks of downloading Jupyter and configuring it to our liking, we can start to work our way through the various tools it has to offer us. Before we get into anything too heavy, let us start out by renaming our Notebook to something a little more memorable. Your Notebook will probably be named something like 'Untitled' at the very top of your Notebook. To rename this, simply click on the name 'Untitled' and type in your preferred name. This will save automatically but if you want you can also just hit the save icon to do this manually. Jupyter autosaves every 120 seconds to a 'checkpoint file' which doesn't alter your main primarily Notebook - this means that any crashes can be entirely recoverable. You can also revert to any given checkpoint by going "File > Revert to Checkpoint".

There are two really important terms we should know about prior to any actual coding in our Notebook. The first is a 'kernel', think of this as the entire block of the Notebook that deals with executing all the code that is contained in the document. The second is a 'cell', each cell is a

container for text to be displayed in the notebook or for code to be executed by the notebook's kernel. A Notebook will have one kernel that executes the code from many cells.

To break this down slightly more. A cell is the backbone of the Notebook. there are two main types of cells that you should know about. The first is a 'code cell' this is where our code goes to be executed by our kernel. When the code is executed, the notebook will display the output below the code cell that generated it. The second is a 'Markdown cell'. This contains text formatted using Markdown and will be where you can add additional information that goes above and beyond simple comments in a .do file. To create a new cell in Notebook simply click on the plus icon next to the save icon on the top panel.

When running Stata in our Jupyter Notebook there is one golden rule to remember because each cell is separate from one another, each time we wish to execute some code within Stata, we have to call upon it prior to any code. As an example, say I wanted to use a specific dataset on my local disk drive, I would first have to add:

%%stata

Prior to adding within the same code cell:

use "G:\Stata data and do\NCDS\Occupation Codes\UKDA-7023-stata9\stata9\ncds2_occupation_coding_father"

This may sound slightly tedious but remember, the reason we do this is because Jupyter Notebook allows us to use multiple programming languages at the same time across the same environment - I can theoretically clean my code up in Stata and then run my data visulisation in Python or R.

Prior to hitting the 'Run' button at the top of our Jupyter Notebook the code looks very similar to that of any .do file:

```
In [ ]: %%stata

        use "G:\Stata data and do\NCDS\Occupation Codes\UKDA-7023-stata9\stata9\ncds2_occupation_coding_father"
```

However, after we click 'Run', both the code and the output are produced on the same Notebook:

```
In [2]: %%stata

        use "G:\Stata data and do\NCDS\Occupation Codes\UKDA-7023-stata9\stata9\ncds2_occupation_coding_father"

        .
        . use "G:\Stata data and do\NCDS\Occupation Codes\UKDA-7023-stata9\stata9\ncds2
        > _occupation_coding_father"

        .

In [ ]:
```

This doesn't look so amazing with a simple 'use' command, but when we start introducing statistical models and visualisations into the mix, the attractiveness of using Jupyter Notebook becomes easier to understand.

Whilst we have so far displayed the code cell and its execution, when writing code - especially code that can be replicated, we also require detailed comments not just on the code itself but on the research process. For this, we use Markdown cells. Say I wanted to look at how Parental Social Class impacts youth employment trajectories. For this, I would need two things, a variable on parental social class and a set of variables on youth employment trajectories. Fortunately for me, the National Childhood Development Study has just the variables I require, though they are across datasets as this is longitudinal data. I'd need to merge the datasets and also rename the unique id as in some datasets it is capitalised, in others, it is not. If I were to simply do all of this without mentioning it, it would be extremely confusing for anyone trying to follow along with my code - if I provided my code at all. This is where Markdown shines:

This is setting my cd initially to the file within my local drive that hosts the occupational codes (NS-SEC, RGSC, and CAMSIS). It then loads this occupational codes dataset into stata and then with the same cd command resets the location to the location of the sweep 16 dataset.

In [3]:
```
%%stata

rename NCDSID ncdsid

merge 1:1 ncdsid using ncds0123
drop _merge
```

```
.
. rename NCDSID ncdsid

.
. merge 1:1 ncdsid using ncds0123

    Result                      Number of obs
    -----------------------------------------
    Not matched                         3,221
        from master                         0  (_merge==1)
        from using                      3,221  (_merge==2)

    Matched                            15,337  (_merge==3)
    -----------------------------------------

. drop _merge

.
```

Using longitudinal data such as the NCDS, every individual participant is given a unique personal identifier. When using datasets from multiple points in time this allows for them to be merged and successfully link data across time to the same person. The unique id in the occupational codes dataset is capitalised however. Before merging the datasets the "NCDSID" has to be renamed to "ncdsid". After this the occupational codes dataset is now merged with the sweep 16 dataset. After this merge has been completed the generated variable merge is dropped.

In [4]:
```
%%stata

cd "G:\Stata data and do\NCDS\NCDS Sweep 23\stata\stata9"

merge 1:1 ncdsid using ncds4
drop _merge
```

```
.
. cd "G:\Stata data and do\NCDS\NCDS Sweep 23\stata\stata9"
G:\Stata data and do\NCDS\NCDS Sweep 23\stata\stata9

.
. merge 1:1 ncdsid using ncds4

    Result                      Number of obs
    -----------------------------------------
    Not matched                         6,021
        from master                     6,021  (_merge==1)
        from using                          0  (_merge==2)

    Matched                            12,537  (_merge==3)
    -----------------------------------------

. drop _merge
```

Through this process of combining code cells and Markdown cells, you know exactly what I am doing, and also exactly why I am doing it. These longer format 'comments' just don't work properly in a .do file environment.

Keyboard Shortcuts:

One of the best tools that Jupyter has to offer is the 'edit' and 'command' modes. As a default when you are working away in your Notebook, you will almost always be in 'edit' mode - this can be seen by the cells being highlighted in green. If however you were to:

ctrl+enter

Whilst in an active cell, you will then enter 'command mode'. Whilst in command mode the cells will turn blue and you will now have access to a whole host of keyboard shortcut commands that will make your life so much easier. Here is a general list of shortcuts that would be good to remember or keep handy as you are writing your Notebook - feel free to practice with them:

- Toggle between edit and command mode with Esc and Enter, respectively.
- Once in command mode:
    - Scroll up and down your cells with your Up and Down keys.
    - Press A or B to insert a new cell above or below the active cell.
    - M will transform the active cell to a Markdown cell.
    - Y will set the active cell to a code cell.
    - D + D (D twice) will delete the active cell.
    - Z will undo cell deletion.
    - Hold Shift and press Up or Down to select multiple cells at once. With multiple cells selected, Shift + M will merge your selection.
    - 1-6 will produce headers of different sizes to format your piece in a more professional manner

For the entirety of the keyboard shortcut options click on the keyboard icon at the top of the page.

More on Markdown

As mentioned previously Markdown cells are simple text editors - they work very similarly to any HTML enviroment. The basics of Markdown are very easy to master, but they do really enchance your Notebook and make it more comparable to a Journal Paper than a .do file. Here are the basics both written in editor mode in a Markdown cell:

```
# This is a level 1 heading
## This is a level 2 heading
### This is a level 3 heading
#### This is a level 4 heading
##### This is a level 5 heading
###### This is a level 6 heading

This is some plain text that forms a paragraph. Add emphasis via **bold** and __bold__, or *italic* and _italic_.

Paragraphs must be separated by an empty line.

* Sometimes we want to include lists.
* Which can be bulleted using asterisks.

1. Lists can also be numbered.
2. If we want order.

[It is possible to include hyperlinks](https://www.taylorswift.com/)

And finally, adding images is easy: ![Alt text](https://news.harvard.edu/wp-content/uploads/2023/07/202307x_swift_1407_AP23198726852529-2048x1152.jpg)
```

and also executed:

This is a level 1 heading

This is a level 2 heading

This is a level 3 heading

This is a level 4 heading

This is a level 5 heading

This is a level 6 heading

This is some plain text that forms a paragraph. Add emphasis via **bold** and **bold**, or *italic* and *italic*.

Paragraphs must be separated by an empty line.

- Sometimes we want to include lists.
- Which can be bulleted using asterisks.

1. Lists can also be numbered.
2. If we want order.

It is possible to include hyperlinks

And finally, adding images is easy:

Jupyter Notebook can even be set up to produce equations as seen in Latex format within the same environment. This turns this Markdown comment:

Jupyter can also display equations!

```
$$\hat{Y} = \hat{beta}_{0} + \sum \limits _{j=1} ^{p} X_{j}\hat{\beta}_{j}$$
```

Into this:

Jupyter can also display equations!

$$\hat{Y} = \hat{beta}_0 + \sum_{j=1}^{p} X_j \hat{\beta}_j$$

As a quick explanation of the syntax to be used in Markdown:

- **$** : All the Math you want to write in the markdown should be inside opening and closing $ symbol in order to be processed as Math
- **\beta** : Creates the symbol *beta*
- **hat{}** : A hat is covered over anything inside the curly braces of \hat{}. E.g. in \hat{**Y**} hat is created over Y and in \hat{**\beta**}_{0}, hat is shown over *beta*
- **_{}** : Creates as subscript, anything inside the curly braces after _. E.g. \hat{\beta}_{**0**} will create beta with a hat and give it a subscript of 0
- **^{}** : (Similar to subscript) Creates as superscript, anything inside the curly braces after ^
- **\sum** : Creates the summation symbol
- **\limits _{} ^{}** : Creates lower and upper limit for the \sum using the subscript and superscript notatio*n*
- *** : Creates horizontal line
- ** ** : Creates space
- **\gamma** : Creates *gamma* symbol
- **\displaystyle** : Forces display mode (*BONUS 3* above)
- **\frac{}{}** : Creates fraction with two curly braces from numerator and denominator
- **
** : Creates line breaks
- **\Bigg** : Helps create parenthesis of big sizes
- **\partial** : Creates partial derivatives symbol
- **\underset**() : To write under a text. E.g. *gamma* **under** *arg min*, instead of a subscript
- **\in** : Creates *belongs to* symbol

All set!

From here you now know the basics of Jupyter Notebook. You should be able to use Stata, write code, execute it, and create detailed well-formated text within one environment. For example, Jupyter Notebook demonstrates all the skills and tools that we have talked about up to this point check out a fantastic example of open science practices by Vernon Gayle and Roxanne Connelly. They have created a Jupyter Notebook for their entire research paper on "An investigation of Social Class Inequalities in General Cognitive Ability in Two British Birth Cohorts".

Sharing Jupyter Notebooks

Jupyter Notebooks offer an excellent avenue for a researcher seeking to expand on their research process. These expansions should not exist in a vacuum, however. Jupyter Notebook offers a great opportunity to engage in open science practices. This can be witnessed 'in house' through the expansion of commenting tools via Markdown, as well as additions of option text, images, videos, websites etc. However, none of this matters if no one can actually easily access the Notebook you have created. This short post demonstrates that Jupyter Notebooks are the tool to use if you wish to pursue open science-based practices.

Jupyter Notebooks are written and saved as an .ipynb file. This is a text-based interactive file - it takes its name from the old-school IPython Notebook Style. These .ipynb files have two attractive qualities. The first is that it can be saved/exported as an HTML file - which can then be easily hosted on any website. The second is that it can also be kept as it is - as a .ipybn file - and hosted on a repository site such as GitHub. Github doesn't simply store the file like a Word document, it displays the entire document as it was last executed and saved. For a demonstration of this, check out the Notebook hosted by Vernon Gayle and Roxanne Connelly here.

The .ipybn file type used by Jupyter Notebook is thus: Human-readable using plain text format (through JSON), has an open standard meaning it is open for anyone to access and use, finally it can be converted to other formats like HTML, PDF etc.

Jupyter Notebook Extensions

To get extensions in your Jupyter Notebook environment you'll first need to install 'Nbextensions'. This can be done through PIP or through Anaconda Prompt depending on how you originally installed Jupyter. For PIP users type:

pip install jupyter_contrib_nbextensions && jupyter contrib nbextension install

For Anaconda users type into the Anaconda Prompt:

conda install -c conda-forge jupyter_contrib_nbextensions

Following this with a refresh of the Jupyter environment you should now see an 'Nbextensions' tab on your homepage environment. Sometimes this doesn't happen, try typing this into the terminal or Anaconda environment:

jupyter nbextensions_configurator enable --user

This should with another refresh show the tab. If it does not and the problem persists check out this github post about other common issues and run down them.

Now we are all set up we can start to explore some of these extensions. The first thing you will notice when clicking on the tab is that there is a lot of possible extensions.

Configurable nbextensions

☐ disable configuration for nbextensions without explicit compatibility (they may break your notebook environment, but can be useful to show for nbextension development)

filter: by description, section, or tags

☐ 2to3 Converter	☐ AddBefore	☐ Autopep8	☐ AutoSaveTime
☐ Autoscroll	☐ Cell Filter	☐ Code Font Size	☐ Code prettify
☐ Codefolding	☐ Codefolding in Editor	☐ CodeMirror mode extensions	☐ Collapsible Headings
☐ Comment/Uncomment Hotkey	☑ contrib_nbextensions_help_item	☐ datestamper	☐ Equation Auto Numbering
☐ ExecuteTime	☐ Execution Dependencies	☐ Exercise	☐ Exercise2
☐ Export Embedded HTML	☐ Freeze	☐ Gist-it	☐ Go to Current Running Cells
☐ Help panel	☐ Hide Header	☐ Hide input	☐ Hide input all
☐ Highlight selected word	☐ highlighter	☐ Hinterland	☐ Initialization cells
☐ isort formatter	☑ jupyter-js-widgets/extension	☑ jupyterlab-plotly/extension	☐ Keyboard shortcut editor
☐ Launch QTConsole	☐ Limit Output	☐ Live Markdown Preview	☐ Load TeX macros
☐ Move selected cells	☐ Navigation-Hotkeys	☑ Nbextensions dashboard tab	☑ Nbextensions edit menu item
☐ nbTranslate	☐ Notify	☐ Printview	☐ Python Markdown
☐ Rubberband	☐ Ruler	☐ Ruler in Editor	☐ Runtools
☐ Scratchpad	☐ ScrollDown	☐ Select CodeMirror Keymap	☐ SKILL Syntax
☐ Skip-Traceback	☐ Snippets	☐ Snippets Menu	☐ spellchecker
☐ Split Cells Notebook	☐ Table of Contents (2)	☐ table_beautifier	☐ Toggle all line numbers
☐ Tree Filter	☐ Variable Inspector	☐ zenmode	

It won't be feasible to go over all of these in one post so I will focus on a few that should be very useful to implement but go through and click on all of them - a brief description .md file and an image of what each extension does should be visible to you.

Move Selected Cells

The move selected cells extension allows you to avoid the need to enter into command mode. With this extension, you can simply use alt + up or alt + down to move cells or groups of cells.

Hinterland

This extension adds an autocomplete feature to Jupyter. This is a great extension for those of you (myself included) who type faster than we think.

Hide Input and Hide Input All

Often times we never want to hide our code. It inhibits open science practices. However, some circumstances can call for hiding some of our code - this may be useful if you are merging lots of datasets together and need to use the 'cd' and 'use' commands in Stata a lot. A comment saying you are doing this, instead of showing every instance of this code would suffice. This extension allows you to hide portions of your cells.

Table of Contents (2)

If the intention in using Jupyter Notebook is to write good code and produce a report able to be replicated, you will end up with a large Notebook. This extension fathers all the headings available in your Markdown cells and shows them in the sidebar, making browsing much easier.

Spellchecker

Another amazing extension if like myself, you type far too quickly. As the name suggests, the spellchecker extension highlights any spelling errors in the Markdown cells.

Collapsible Headings

This extension is similar to the way headings work in Word. It simply adds the ability to collapse everything within that Mardown heading. The collapsed status of the headers is also saved in the cell metadata and reloaded when you reload the Notebook.

AutoSaveTime

This extension allows you to change how often your Notebook autosaves. If you are conscious of crashes etc then you can shorten them, or if you are running this instance on a weaker machine, elongate the autosave.

Execute Time

After executing a code cell a stamp will be placed by the cell on the date and time executed as well as how long it took for the code to execute. This is great for open science practices (combatting P-Hacking, HARKing etc) as well as helping you improve efficiency on your code.

nbTranslate

This extension is great when you are reading other people's Notebooks and they are using a different language to your own. It translates comments for you.

Highlighter

This extension adds the ability to highlight Markdown comments within your Notebook.

Beyond Extensions

Beyond the extensions used in nbextensions two more vital 'additions' to Jupyter should be mentioned. The first is integrated into Jupyter already - multicursor support. Jupyter supports multiple cursors. Simply click and drag your mouse while holding down Alt. This is exceptionally useful for researchers engaging in occupational data that have to manually recode SOC codes etc.

The second and most powerful of all extensions is probably 'RISE'. RISE is an extension that allows you to convert in real-time your entire (or individual sections if you want) Jupyter Notebook into a PowerPoint-like presentation. To use RISE you will first need to install it, again either via PIP:

pip install RISE

or via Anaconda Prompt:

conda install -c conda-forge rise

RISE should then be automatically selected in your nbextensions configuration.

Note for Older macOS (Sierra 10.12 or older)

If you are using a very old macOS you will run into lots of issues when attempting to add extensions to your Jupyter Notebook. This will primarily come from an issue with PIP rather than the extensions themselves. This may be overcome by installing:

curl https://bootstrap.pypa.io/get-pip.py | python

Which uninstalls PIP and reinstalls it to the most up-to-date version - then you can proceed with downloading the extensions. Sometimes this just does not work, however. I have yet to find a solution to this.