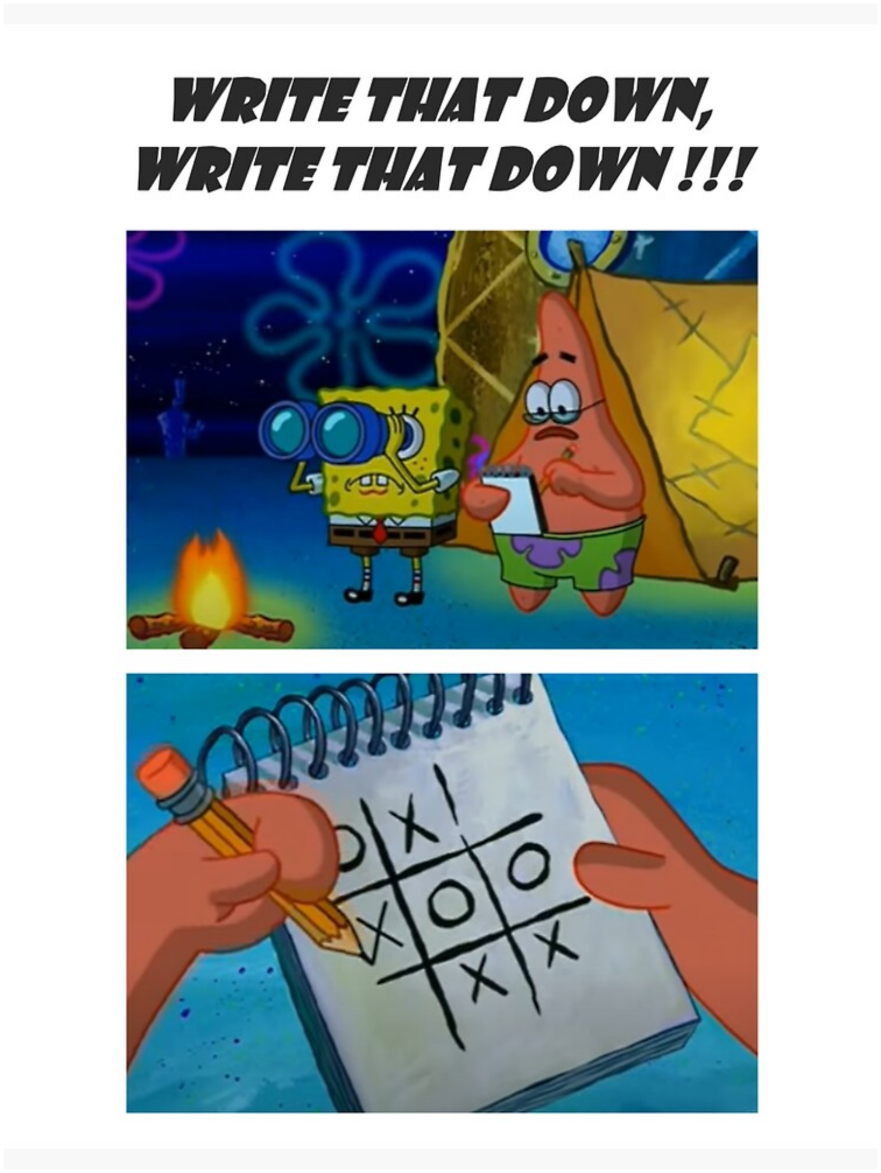
Don't Do That by Hand!!!
A Coding Guide to the Interactions Homework & How to Use Jupyter Notebooks

Christopher Valdes & Gia Jadick

October 15, 2025



1 Introduction

Welcome to the first lunch of the *Research Skills for Medical Physics* seminar series! Today, we are going to take a look at how to use Jupyter notebooks to make your homeworks beautiful and to delight your TAs, with a focus on the upcoming *Interactions* homework.

As you know, the homework for this class has a lot of repetitive calculations, and while repetitive reinforcement is a thing, there comes a point where an error is likely to reoccur. As a result, we will go over how to use Python functions and how to code equations for a future homework (charged particle interactions) so you don't have to re-write each calculation by hand. As your *Interactions* TA, I (Chris) will gladly accept PDF documents of your code directly from a Jupyter notebook.

2 What You Will Need

You will need Jupyter, which you may have already installed for the coding bootcamp. This comes with everything you need for creating Jupyter notebooks (ipynb files) as well as the `nbconvert` command that we will get to later on for exporting to PDF.

The PDF conversion also requires a LaTeX distribution like `xelatex` or `mactex`. The process is different if you have Windows or Mac.

2.1 Installing Jupyter

Chris and I (Gia) have incredibly different philosophies on the best way to install Jupyter. Chris argues that it is easiest to install Anaconda Navigator from browser, since it comes with everything you might need. I argue that this takes ten million years to download and adds unnecessary bloat to your computer, taking up precious storage space that you will undoubtedly need later, so I use a minimalist install. Here are our two methods:

2.1.1 Chris's Method: Anaconda Navigator

Downloading Anaconda Navigator from the link below installs Jupyter notebook alongside other software such as Jupyter Lab, Spyder, and more! The benefit of installing Anaconda is that it automatically installs Python dependencies and several other packages/applications you may need for coding. The con is it may clash with other Python package managers and you might not use everything. Link:

https://www.anaconda.com/download

From this link, you can create an account and download the "Distribution Software" for Mac, Windows, or Linux. Simply follow the instructions for application installation, and eventually, it'll completely install. Once installed, open Anaconda Navigator and wait for it to load, then scroll down to wherever Jupyter notebook is, and click "Open". It may ask you to choose a software to run notebook on; if so, any web browser will suffice.

2.1.2 Gia's Method: Just Jupyter

If you already have Python (Macs come with it pre-installed, Windows you can download it online), you can just use `pip`. In terminal, simply run:

```
pip install jupyter
```

Jupyter is a relatively big package but still should install within a few minutes. Better yet, if you have `uv` (the Rust-compiled Python package manager) this can take *just a few seconds* with `uv pip install jupyter` instead.

3 LaTeX distribution for PDF conversion

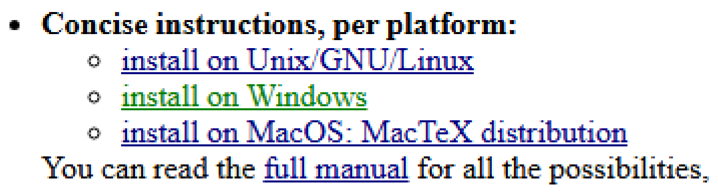
In addition to Jupyter, a LaTeX distribution is necessary to render the markdown cells interspersed throughout your ipynb into nice PDFs. There are a variety of options that will work. For Windows, here are instructions for `xelatex`, and for MacOS, `mactex`.

3.1 Windows

If you have a Windows laptop, go to the following link to install `xelatex`:

https://www.tug.org/texlive/

Click on the link that says "install on Windows" (in green), as below:



- **Concise instructions, per platform:**
  - install on Unix/GNU/Linux
  - install on Windows
  - install on MacOS: MacTeX distribution

  You can read the full manual for all the possibilities.

Then click on "install-tl-windows.exe" under the section titled "Easy Install", as seen in the figure below.

install-tl-windows.exe.

It may install a large folder of strange files instead of automatically running the installation code. If this happens, don't worry! Simply download the folder, unzip it (if necessary), navigate inside the folder to the installation application (double click this file to run it).

3.2 MacOS

If you have a Mac user, you can either use the same method as above (instead clicking "install on MacOS" *or* you have an easier and elegant alternative to use the Mac package manager `brew`. Simply go to the terminal and run:

```
brew install --cask mactex
```

This includes `xelatex` and everything else Jupyter `nbconvert` might need for PDF.

3.3 Installation *Might* Take Time

Regardless of your device, the installation process can take a long time in order to properly install. Make sure to run the installation now so that you can start converting notebooks to PDF before the end of this session!

4 Let's Start Coding!

In Python, functions are blocks of code that you can call to repeatedly perform a certain task. For example, let's take a look at a very simple version of a function.

```
import numpy as np
def average(x, y):
    combined = np.array(x + y)
    z = np.mean(combined)
    return z
```

We define a function by typing def and giving it the name *average* with parentheses. In parentheses, we establish the variables x and y for the function to use. In the context of this problem, we have two lines of code that calculates the mean of the variables x and y and assigns the result to the variable z. However, x and y are entered as lists and not arrays, so the function will convert the variables into a single array (which is what the variable *combined* does. Finally, the last line returns the value we want to output from the function, which is z. Keep in mind that the *combined* variable is a local variable specific to this function. Outside of the function, you cannot reference this variable unless you return it. To run this function, we therefore have:

```
x = [3, 5, 6]
y = [1, 2, 8]
z = average(x, y)
```

Keep in mind that your variables do NOT have to exactly match the variables you use when you define your function.

```
group1 = [3, 5, 6]
group2 = [1, 2, 8]
mean = average(group1, group2)
```

We can also return multiple values from the same function. For example, in the following code snippet, we use the same function *average*, but we are returning both mean and max values, thus slightly changing our function to accomplish this.

```
import numpy as np
def average(x, y):
    combined = np.array(x + y)
    z = np.mean(combined)
```

```
        a = np.max(combined)
        return z, a
    group1 = [3, 5, 6]
    group2 = [1, 2, 8]
    mean, maximum = average(group1, group2)
```

However, functions do not necessarily need to return a numerical value. They can return any type of variable you define, such as strings!

```
    def print_string(input):
        statement = "Inputed: " + str(input)
        return statement
```

In the context of the function above, we input a variable, and the function converts it to a string to return an output string of the defined statement. We can also take advantage of this by putting these functions into a loop! As an example, the following snippet of code allows us to print a custom string without doing it manually.

```
    for i in range(5):
        phrase = print_string(i+1)
        print(phrase)
```

Overall, we can use functions to redo the same task repeatedly without having to rewrite the equation! Rewriting or recoding the same task or equation can lead to inconsistencies between answers, but by using functions, we can ensure that all of our calculations are consistent!

5 Example for *Interactions*

Next week, we will cover charged particle interactions in *Interactions*. (Hooray!) The homework for these particular lectures have very repetitive calculations that are very easy to mess up when you input them into code or a calculator manually (Sagada is still traumatized by this). Therefore, I highly recommend coding this particular assignment instead. Let's take a look at one of the equations you will need!

$$\left(\frac{dT_s}{\rho dx}\right)_c = \frac{2Cm_oc^2z^2}{\beta}\left[ln\left(\frac{2m_oc^2\beta^2H}{I^2(1-\beta^2)}\right) - \beta^2\right] \tag{1}$$

Equation 1 is the Bethe soft collision formula (Equation 8.6 in the textbook). I'm not going to explain the equation since I don't want to spoil next week's lectures, so for now, let's try to code it in Python! To make your lives easier, I'll substitute the variable k for the first term of the equation, giving us:

$$k = \frac{2Cm_oc^2z^2}{\beta} \tag{2}$$

Leading us to:

$$\left(\frac{dT_s}{\rho dx}\right)_c = k\left[ln\left(\frac{2m_oc^2\beta^2H}{I^2(1-\beta^2)}\right) - \beta^2\right] \tag{3}$$

Using Jupyter notebook, calculate and print the solution with the following variables. Use a function to define the computation needed for Equation 3.

$$C = 0.9\ \frac{cm^2}{g}$$

$$m_oc^2 = 3000\ MeV$$

$$\beta = 0.8$$

$$I = 400\ eV$$

$$k = 0.5\ \frac{MeV}{g/cm^2}$$

$$H = 100\ eV$$

Once you have this solution, recalculate the solution with these variables instead:

$$C = 0.4\ \frac{cm^2}{g}$$

$$m_oc^2 = 1800\ MeV$$

$$\beta = 0.6$$

$$I = 512\ eV$$

$$k = 0.2\ \frac{MeV}{g/cm^2}$$

$$H = 100\ eV$$

6 Why Jupyter? Using Code *and* Text

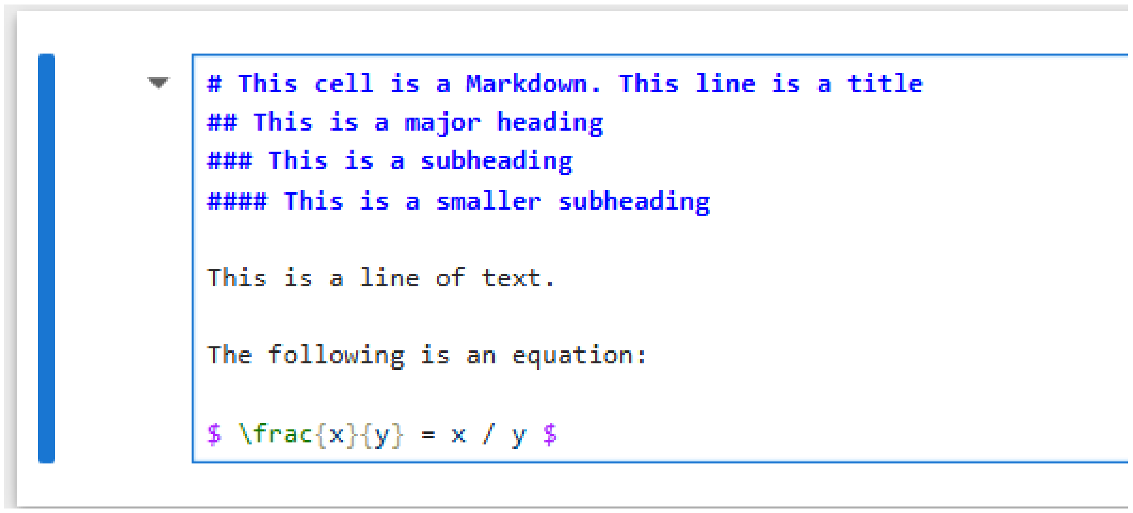
In a Jupyter notebook, you can easily switch back and forth between snippets of text (including equations!) and code. For the sake of my sanity, PLEASE do not just upload a Jupyter notebook full of ONLY code. At least explain your thought process behind your functions and calculations so that I don't have to dissect everything you type out! Now that we already have a Jupyter notebook, let's talk about markdown versus code cells to organize everything into a document that will be easy for me to read.

6.1 Markdown Cells

In Jupyter notebooks, markdown cells are sections of the document that allow you to write text and equations with nice formatting, while code cells are actual blocks of code that will run and show outputs below it (like plots and print statements). Let's talk about the image below! The first first block is a markdown cell, which allows us to type out text and equations. Titles, sections, italics, boldface, and more can be nicely formatted using markdown. Here is a link to the official "cheat sheet" for markdown formatting:

https://www.markdownguide.org/cheat-sheet/

You can reference this as you use markdown, but for now, here is an example of a markdown text you could enter:



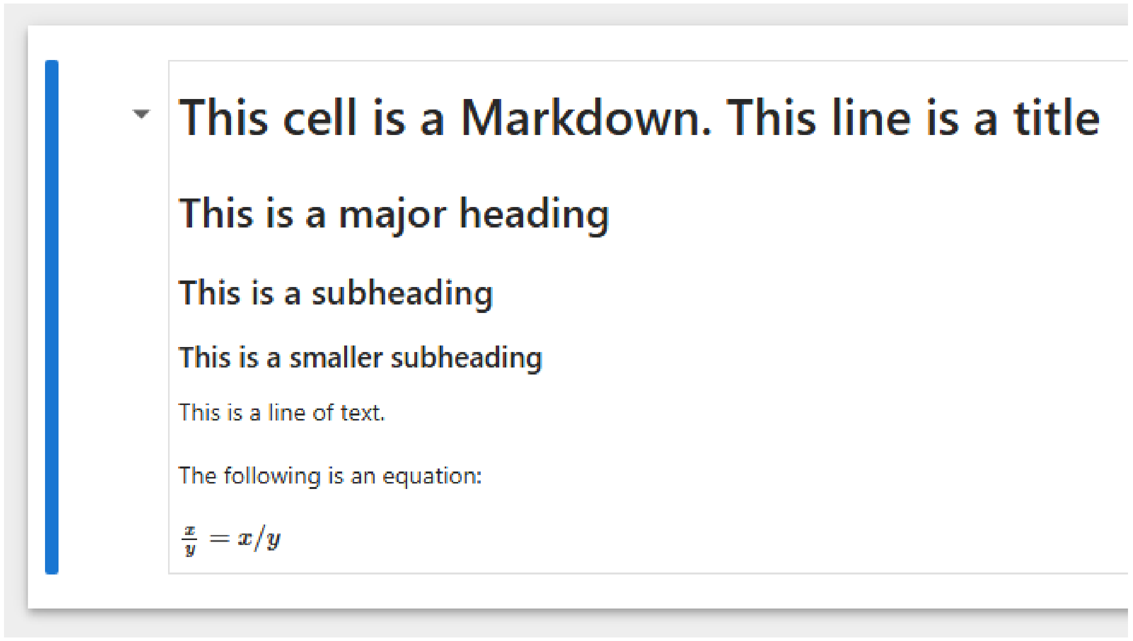
```
# This cell is a Markdown. This line is a title
## This is a major heading
### This is a subheading
#### This is a smaller subheading

This is a line of text.

The following is an equation:

$ \frac{x}{y} = x / y $
```

And when you do CRTL/CMD + Enter on this cell, you get:



This cell is a Markdown. This line is a title

This is a major heading

This is a subheading

This is a smaller subheading

This is a line of text.

The following is an equation:

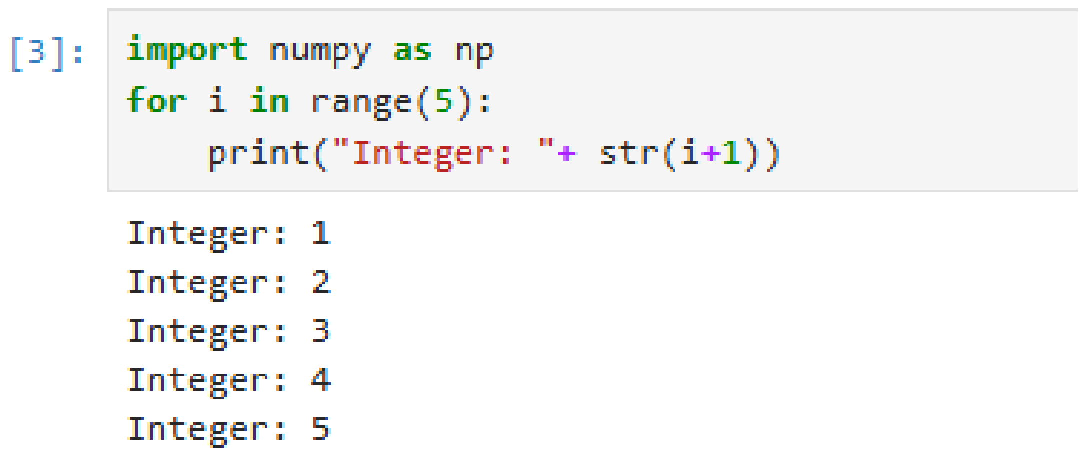
$\frac{x}{y} = x/y$

As you can see, the raw text is rendered with the target formatting. For equations, you use

LaTeX format. Instead of the `\begin{equation}` formatting you might typically use, make sure to use single or double dollar-signs `$$` to book-end your math (depending on whether you want in-line math or whole-line equations). If you have used LaTeX before, then great! You know what to do. If not, don't worry! I was in your shoes when I (Chris) started using notebook for the first time in my first year of the GPMP. You can just Google/ChatGPT the formatting to help you. Regardless, feel free to play around with markdown formatting to match what looks nice to you! As long as I can read it in your assignments, that's all I, as your TA/Supreme Lord Emperor, care about.

6.2 Code Cells

Now let's talk about coding in Jupyter notebook. When you click to create a new cell, you can select the dropbar at the top of your screen and click "code" to designate a chunk of code. You can easily convert cells between code, markdown, and raw formats this way. Each time you run a chunk of code, the output is placed immediately below the cell (as seen below).



```
[3]: import numpy as np
     for i in range(5):
         print("Integer: "+ str(i+1))

     Integer: 1
     Integer: 2
     Integer: 3
     Integer: 4
     Integer: 5
```

7 Exporting to PDF

Once you finish your markdown, it's now time to save your document as a PDF! You could always File/Print the notebook page, but the formatting will be horrendous, and it may not accurately display everything. Fortunately, Jupyter comes with `nbconvert` that can render your ipynb file in a variety of export formats that are much more reader-friendly. With our previous LaTeX distribution install, we are ready to convert to PDF.

7.1 Command-line interface

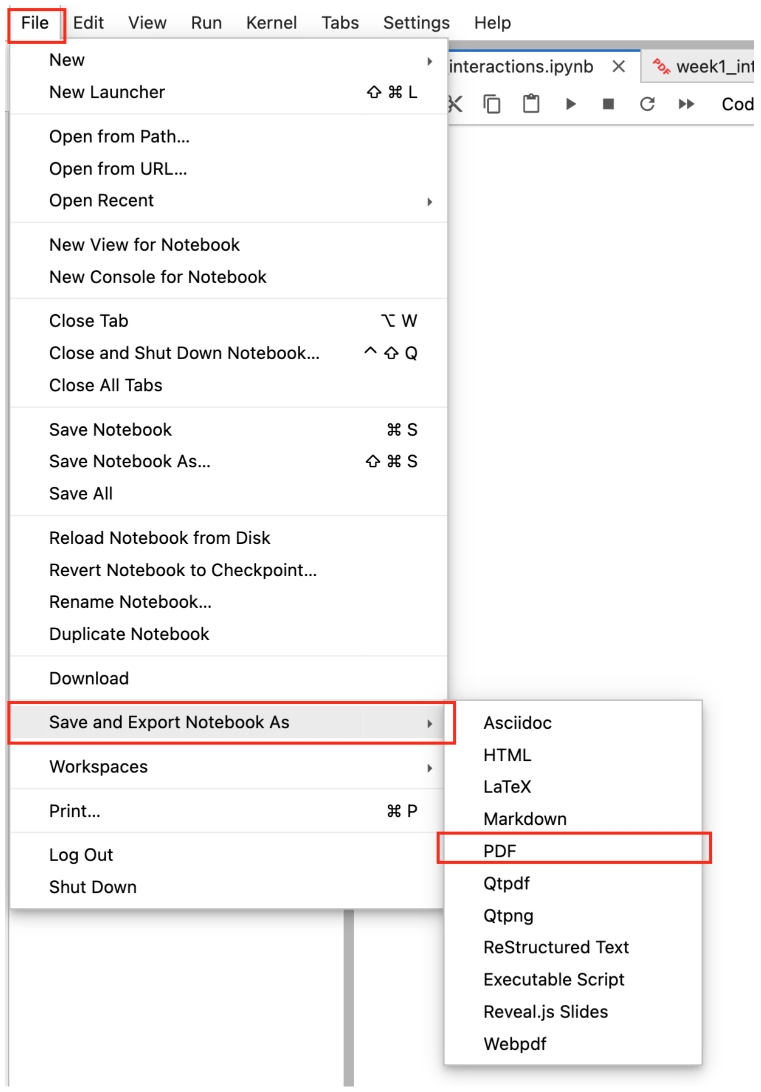
In terminal, navigate to the directory containing your Jupyter notebook file (say, "example.ipynb") and run:

```
jupyter nbconvert example.ipynb --to pdf
```

And POOF! You have your PDF document saved as "example.pdf"!

7.2 Jupyter Lab

Your LaTeX install should also make it possible to convert to PDF straight within Jupyter lab. I think this is just a GUI wrapper for `nbconvert`, but it is nice if you already have the ipynb open in Jupyter lab. Just click File > Save and Export Notebook As > PDF, and it should render just like above!



8 Conclusion

I hope you find this guide and coding exercise useful for future assignments and research! Moving forward, many supplemental code for coursework and research will use Jupyter notebooks. Also, it's extremely useful to use LaTeX to write your proposal and dissertation, so continue practicing using Jupyter notebook to prepare you for it. Let us know if you have any questions!

— *Chris & Gia*