tutorial001

January 9, 2022

Hello! This is a short introduction for using JupyterLab, which is a really cool tool to both analyze data and share your methods and findings with others. (Note. We are also going to learn a little bit of Python programming!) Introduction to the JupyterLab App for Data Science Hello! This is a short introduction for using JupyterLab, which is a really cool tool to both analyze data and share your methods and findings with others. (Note. We are also going to learn a little bit of Python programming!) Let's prepare our programming environment To start the fun, first, we have to install the software.

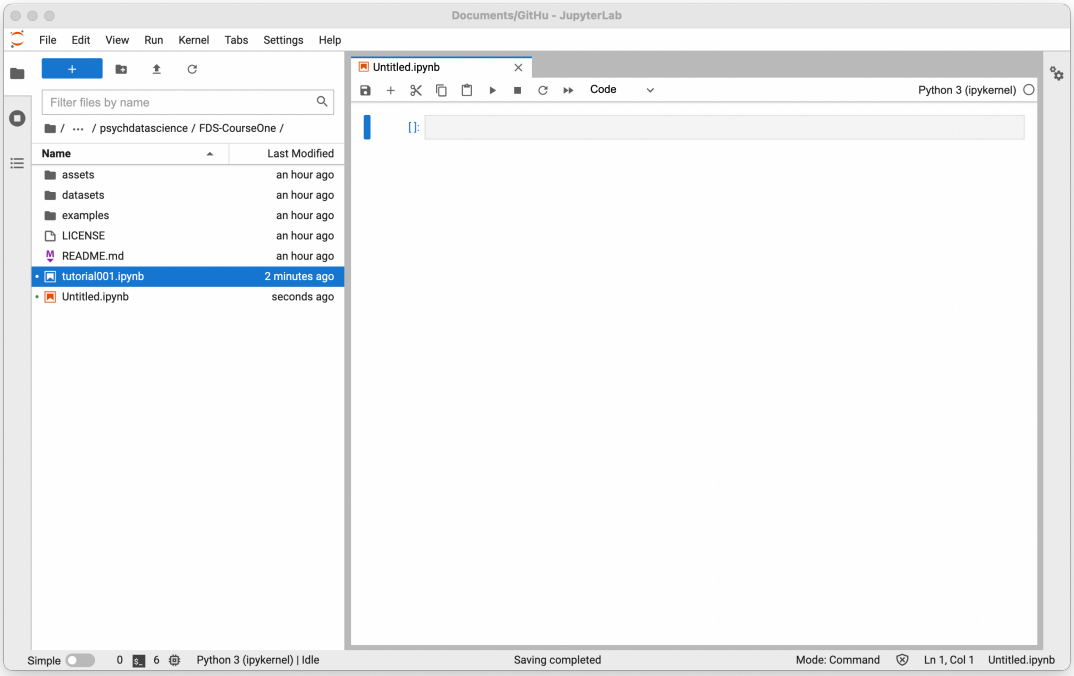
To start off, we are going to use the JupyterLab App, which is an all-in-one standalone application for using Python to do data science. So we don't need to worry about installing Python and various Python libraries separately. We just install the JupyterLab App and we're done.

To install the JupyterLab App, go here https://github.com/jupyterlab/jupyterlab-desktop#download and follow your nose.

The files (i.e. documents) we will be working with are called "notebooks" and have the file extension "ipynb".

Note: As just mentioned, the files we'll be working with are called notebooks. Originally, the program used to create and edit these files was "Jupyter Notebook" (and you can still use it). We will be using the JupyterLab App, however, because it's better. *Whenever we say "notebook", we are referring to the files, not the old software.*

Go ahead and launch the JupyterLab App. Something like this will then appear:



This window is pretty typical of an Integrated Development Environment (or IDE) for programming. If you have used RStudio in the past, for example, it should look a little familiar. Like with all IDEs, there are different "panes" or areas for doing different things.
One thing that is different and cool about JupyterLab is that your code, the output of your code, and descriptions of what you are doing and why are all interleaved into one human-readable document – a "notebook". In fact, you are looking at a notebook right now, one that has been exported to a PDF file. The large pane on the right is a blank notebook just waiting for you!

0.0.1 Our first code!

At the top of your new notebook, you should have a gray bar with a [ ]: to its left. This is a **code cell** which - *surprise* - is where we enter code. Try typing `print("hello world!")` into it, and then hit **Shift-Return** or **Shift-Enter** to run it. You should get:

```
[1]: print("Hello world!")
```

```
Hello world!
```

Notice that your notebook has its own little toolbar at the top. It might be a good time to save your notebook by hitting the little disc icon.
Also notice that when you hit Shift-Enter a new empty code cell was created below the first one, and your cursor is already in it ready to write more code!
Notice also that there is a fat blue vertical bar to the left. This blue bar always denotes the current cell. To change the current cell, just click the cell you want to make current, and then you can edit that cell.

0.0.2 Some preliminaries; we are going to get data-science-specific stuff we need.

One of the things that makes Python so awesome for data science is that has been extended with a bunch of add-ons called libraries.
Type the following two lines into your new code cell and hit Shift-Enter (you can also hit the little run/play button in the notebook toolbar, but I think you'll find hitting Shift-Enter easier).

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

The nicknames after "as" are just to save us some typing later on. For example, to make a plot, we will be able to type "plt.plot()" instead of "matplotlib.pyplot.plot()"!
Libraries we will be using a lot and their traditional nicknames are:

- numpy as np
- pandas as pd
- matplotlib as mpl
- matplotlib.pyplot as plt

As a free human being, you can use whatever nicknames you want, but it is **strongly** recommended that you stick with the traditional ones, as they have become part of the de facto Python data science community language! And for the purposes of this class, please stick to these nicknames.

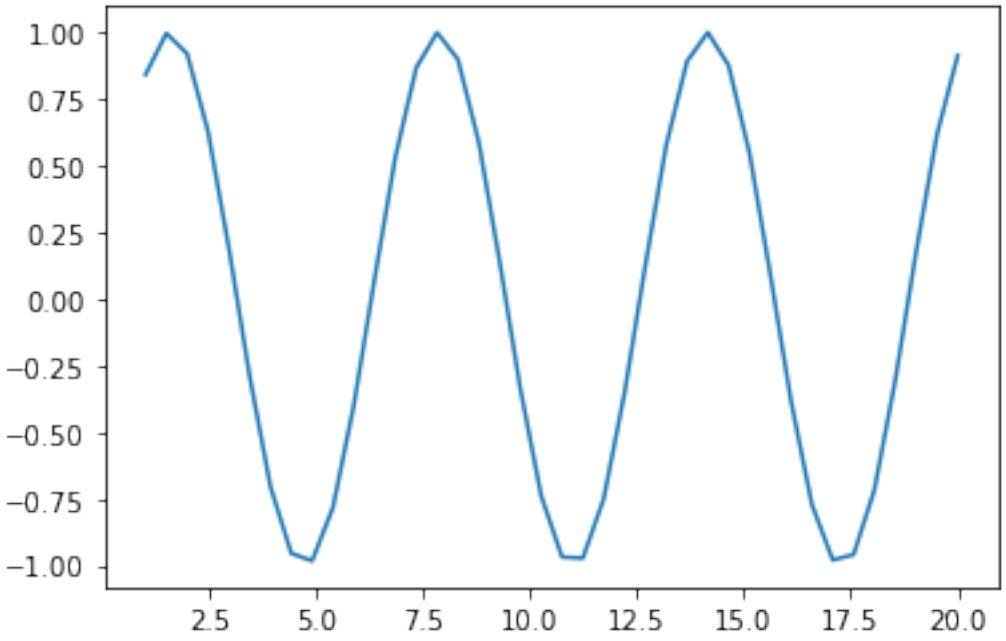
We will also be using scipy a lot, but scipy is huge, so it's best not to import the whole thing but rather to import the specific submodules you need (we'll do that later on).

0.0.3 Okay, let's plot something!

Let's type the following in and hit Shift-Enter to see how easy it is to make a simple plot:

[3]:
```
x = np.linspace(1, 20, 40)
y = np.sin(x)
plt.plot(x, y)
```

[3]:
```
[<matplotlib.lines.Line2D at 0x7fd218847fd0>]
```



From now on, we're going to take the **Shift-Enter** for granted - just do that after you finish entering anything into a cell. Also, anything that's in a gray box that runs across the page is code you should enter in your notebook:

[4]:
```
print("This is code you should enter and hit Shift-Return!")
```

```
This is code you should enter and hit Shift-Return!
```

So let's look at the code to plot that sine wave! The first thing you probably noticed and thought was weird was the way functions are called (e.g. `np.sin()` instead of just `sin()`. In R for example, if we wanted to make a ggplot, we would `library(tidyverse)` early in our code and then, when we wanted to make our plot, we would just call the ggplot function: `ggplot(mydata, aes(...)...)`. In Python, we have to use the name of the package (library), *or the nickname we gave it*, as a prefix every time we call the function. So this:

```
import numpy
x = linspace(1, 20, 40)
```

Would give an error; Python would complain it didn't know about `linspace()`. Instead, we could do this:

```
import numpy
x = numpy.linspace(1, 20, 40)
```

which does make it clear that `linspace()` comes from `numpy`, but we could save ourselves some typing and, perhaps more importantly, space in our lines of code, by assigning the standard nickname, `np`, when we import it:

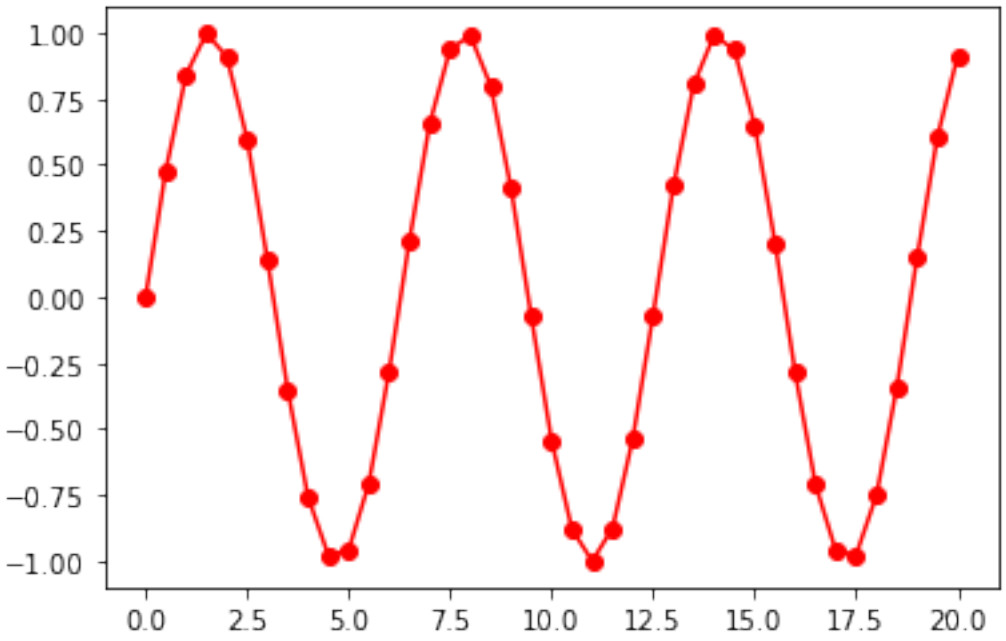
```
import numpy as np
x = np.linspace(1, 20, 40)
```

which, if we all stick to the standard nicknames, makes it clear that `linspace()` comes from `numpy` and saves us some typing and space.

Now let's look at all 3 lines of code (with a little tweek) and the output, and then we can see what each line does. (Go ahead and enter this code in a new cell yourself and run it – respect the gray box!)

[5]:
```
x = np.linspace(0, 20, 41)
y = np.sin(x)
plt.plot(x, y, 'ro-')
```

[5]:
```
[<matplotlib.lines.Line2D at 0x7fd2007568e0>]
```



In the first line, np.linspace() makes a linearly spaced set of 41 numbers running from 0 to 40. We can make the call to `np.linspace()` more clear by assigning the starting point, stopping point, and number of numbers desired to *variables*, and then using those variables as inputs to np.linspace(). In Python, we are allowed to assign values to multiple variables on a single line. So we'll assign a starting point, a stopping point, and a number of numbers, and then use np.linspace() to create that set of numbers:

[6]:
```
start, stop, n = 0, 5, 6
x2 = np.linspace(start, stop, n)
print(x2)
```

```
[0. 1. 2. 3. 4. 5.]
```

So now it's obvious that `x = np.linspace(start, stop, n)` gives us `n` values from `start` to `stop` (inclusive), and then assigns those values to the variable on left of the equals sign. If we look back

at the plot just above, we can see that the x axis runs from 0 to 20, and the plot itself has about 10 points between 0 and 5, or about 40 total (41 to be exact).

Note if you are brand new to programming: a *variable* is just a name for a number or set of numbers. So instead of having to remember or figure out what "5" is, we give it the name "stop" to make our lives easier.

The line `y = np.sin(x)` is pretty self-explanatory. It takes the sine of each element of `x` and assigns them to corresponding elements of `y` (creating the variable `y` for you in the process).

[7]:
```
y2 = np.sin(x2)
print(y2)
```

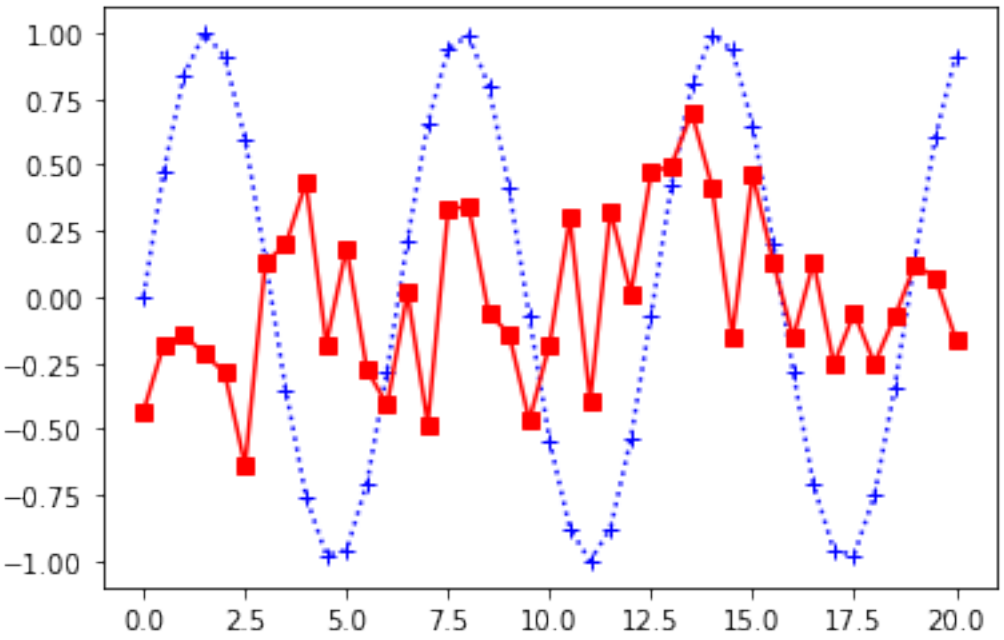
```
[ 0.          0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427]
```

Now for the fun one. The line `plt.plot()` is the quick & easy way to make plots. In its simplest form, you just pass it x and y values and let it do its thing (like for the first figure, above). For the second plot, we also passed it a string of the form **'[color][marker][line]'** to customize the plot.

Let's make another plot.

[8]:
```
start, stop, n = 0, 20, 41
x = np.linspace(start, stop, n)
y1 = np.sin(x)
y2 = 0.3*np.random.randn(n) # Gaussian noise from numpy.random (scaled down by␣
↪0.3)
plt.plot(x, y1, 'b+:')
plt.plot(x, y2, 'rs-')
```

[8]:
```
[<matplotlib.lines.Line2D at 0x7fd20a2315b0>]
```



Notice the `# Gaussian noise...` in the 4th line. Anything after a `#` in Python is a comment, and is for human consumption only.

Look here for more of what `matplotlib.pyplot.plot()` can do!

A couple more things before we call it a day. Notice that we've used `print()` a lot to examine variables. This is useful. Forget what `n` is?

[9]:
```
print(n)
```

```
41
```

Forget what `x` is?

[10]:
```
print(x)
```

```
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5  6.   6.5
  7.   7.5  8.   8.5  9.   9.5 10.  10.5 11.  11.5 12.  12.5 13.  13.5
 14.  14.5 15.  15.5 16.  16.5 17.  17.5 18.  18.5 19.  19.5 20. ]
```

That's fine, but could get cumbersome if `x` was large. So maybe we could just peek at the first, say, 5 values. We can. In fact, we can peek at any contiguous subset of values easily by *indexing* into `x`. For example:

[11]:
```
print(x[0:5]) # print the first 5 elements
```

```
[0.  0.5 1.  1.5 2. ]
```

[12]:
```
print(x[-5:]) # print the last 5
```

```
[18.  18.5 19.  19.5 20. ]
```

[13]:
```
print(x[5:10]) # print the 6th through 10th
```

```
[2.5 3.  3.5 4.  4.5]
```

Note that these things are "**zero based**". That is, the first element is element number zero:

[14]:
```
print(x[0]) # the first number
```

```
0.0
```

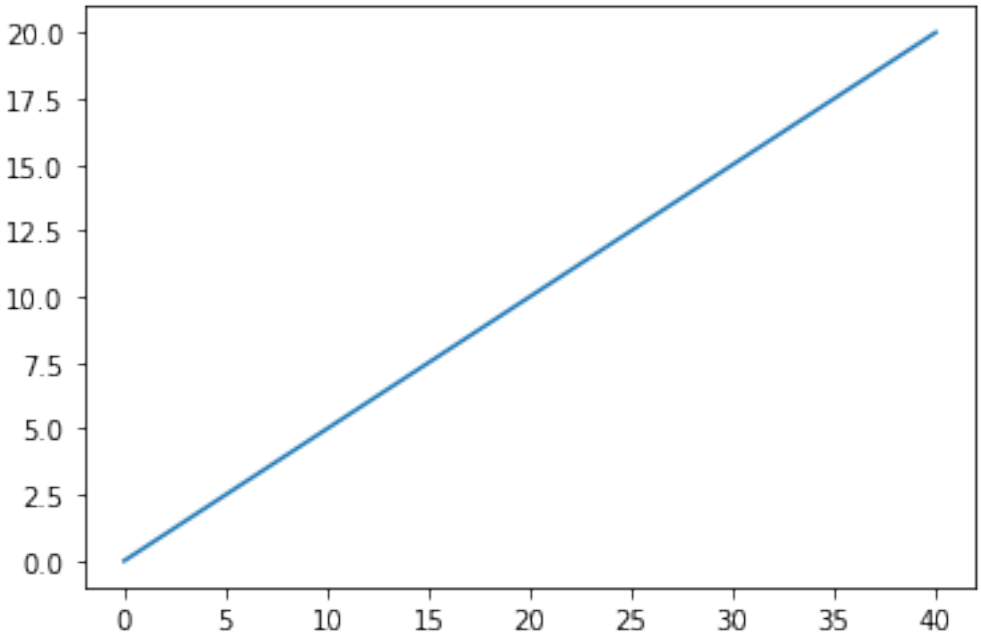
[15]:
```
print(x[1]) # the second number
```

```
0.5
```

We can also just plot x – this will plot the value of x against its index (which in this case should be the line y=x):

[16]:
```
plt.plot(x)
```

[16]:
```
[<matplotlib.lines.Line2D at 0x7fd1f8236d60>]
```



But what if we have now forgotten what all variables we have? We can take a quick look this way:

[17]:
```
%who
```

```
n        np      plt     start   stop    x       x2      y       y1
y2
```

Or we can take a more comprehensive look like this:

[18]:
```
%whos
```

```
Variable   Type       Data/Info
------------------------------
n          int        41
np         module     <module 'numpy' from '/Us<…>kages/numpy/__init__.py'>
plt        module     <module 'matplotlib.pyplo<…>es/matplotlib/pyplot.py'>
start      int        0
stop       int        20
x          ndarray    41: 41 elems, type `float64`, 328 bytes
x2         ndarray    6: 6 elems, type `float64`, 48 bytes
y          ndarray    41: 41 elems, type `float64`, 328 bytes
y1         ndarray    41: 41 elems, type `float64`, 328 bytes
y2         ndarray    41: 41 elems, type `float64`, 328 bytes
```

Note that this note only gives us information about our variables, but also tells us which modules we have imported (np = numpy and plt = matplotlib.pyplot).