0a_intro_to_jupyter

March 19, 2021

1 Intro to Jupyter notebooks

Sources

- This notebook is closely based on a notebook by Todd M. Gureckis introducing jupyter notebooks for psychology undergrads.
- Several of the sections on how to use the Jupyter Interface were developed and shared by Jessica Hamrick as part of a course developed at UC Berkeley.
- The sections on Markdown also include materials based on this blog post

The "notebook" interface is an approach to a programming, data analysis, computational modeling etc. that combines code with text, images, and other multimedia in the same document.

- The idea for computational notebooks has been around for a long time (e.g., Mathematica is a commercial product that has long provided a notebook interface for data analysis and programming).

However, Jupyter is now one of the leading tools that scientists, including computational (psycho)linguists and cognitive scientists, use to write code, analyze their data, model the results.

Reasons for Jupyter's popularity:

1. Jupyter is open source ('free as in speech') and free ('free as in beer'), just like Python
   - it is not owned by any one person or company
   - scientists and other programmers devote their time to improving Jupyter
   - the code is readable by anyone, which means it is easier to catch bugs
   - there is a large, thriving user base, so it's reliable
2. Jupyter works with several different programming languages including R, Python, Matlab and others
3. Jupyter runs in a standard web browser.
4. Jupyter has many extensions that provide interactive widgets, animations, beautiful graphics, etc.
5. Companies use Jupyter internally, and are looking to hire people with this skill.

Pretty much every lecture of this Advanced Psycholinguistics (Ling 158) course will be a jupyter notebook.

1.1 The "notebook" and "kernel" concepts

A Jupyter **computational notebook** is a document-based approach to structuring programming, modeling, data analysis etc. It is a form of literate programming.

We will use the "classic" Jupyter Notebook interface, which consists of a single document that you interact with similar to the way you would interact with a word processor.

- There is a newer interface called JupyterLab which looks more like RStudio's or Matlab's interface allowing multiple windows and views of the running code.
- We use the traditional interface here because it's simple and faciliates reading along.

A computational notebook is made up of a set of elements called **cells** that group little bits of information together. There are two main kinds of cells:

- text cells, specifically, Markdown cells; we'll come back to Markdown
- code cells, in our case, Python code cells

The file extension of a notebook document has the `.ipynb` file extension.

- `.ipynb` for 'IPython Notebook', the original version of Jupyter notebooks developed as a particular kind of interface for IPython

A notebook is a list of cells arranged from the top of the document to the bottom.

- cells can contain several types of information, including code, text, images and so forth.

You develop your project by creating new cells, entering information, reordering cells, and saving the results.

The key element which makes a notebook "computational" is that it can be linked to a computing engine known as a "kernel", which can run on the same computer or another computer using the Internet.

- The kernel is separate from the notebook and can be stopped and started independently from the program displaying the notebook itself
- The magic happens when a code cell is "executed"
  - you can also "execute" a text cell, which just renders the text from Markdown annotation to something easily readable by a human
- When the code from the cell is executed, it is sent to the computational kernel, which runs the sequence of commands contained within the cell
- The results of running the code, e.g., figures, printed messages, etc., are captured and sent back to the notebook from the kernel, where there will be inserted beneath the cell which was just executed

Example text *cell*.

[1]:
```
# example code cell
2 + 2
```

[1]:
```
4
```

1.2 The Notebook State and Kernel State are not the Same

The notebook is simply a way to organize information and the order of information in a notebook might be different from the order in which code has been executed on the kernel.

For example, I can execute a code cell at the end of the document first, and then go back to the beginning of the document and run a cell there. This happens often when you develop your project.

Thus, the order of the cells in the notebook document is really just for presentation, and it is up to you to "run/execute" the cells in whatever order you want.

- for example, if you delete a cell after running it, it doesn't appear in the notebook anymore but its code has not been "undone" on the kernel
- the kernel doesn't know anything about any edits you make to the notebook interface; all it does is receive code when you send it via the "execute cell" command, runs the code, sends back the outputs, and then it waits

This can mean that the current state of the kernel can be different from what your cells or notebook look like, and the outputs captured in the notebook might be obsolete.

If things get out of control, you can simply restart the kernel, which will erase its current state and set it back to a fresh start. You then need to re-execute all of the cells in your notebook to get back to where you were.

This is like telling your friend how to cook a meal over the phone. You have the instructions in front of you (your notebook) and you tell your friend (the kernel) what to do. If you read the instructions out of order, your friend can't tell the difference. They just know – and follow – the steps in the order in which you communicated them.

For your final project, please make sure the notebook runs correctly in the linear order of the cells from top to bottom on a fresh (restarted) kernel. You ensure this by selecting **Restart & Run All** from the **Kernel** menu above, and then carefully checking all the outputs from beginning to end.

You **must** do this before submitting your homeworks or your final project for the class

1.3 Overview of the Jupyter Notebook

We will now go over the basics of the Jupyter interface. Make sure you open this notebook on Google Colab (or on a local installation of Jupyter and Python3) and execute cells, experiment etc. after the class meeting is over.

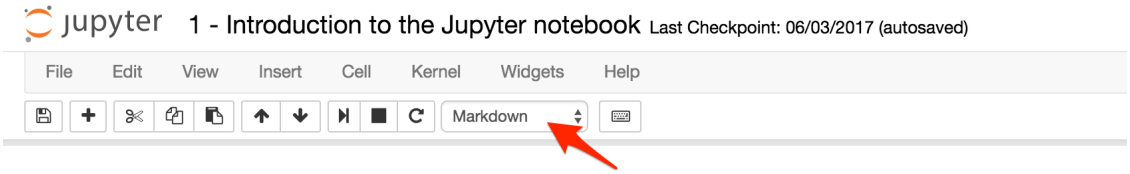
Recall notebooks are made up of a series of *cells*, and the two main types of cells are text cells, like this one, or code cells, like the one below.

- there are 2 other types of cell, but we don't need to talk about them

[5]:
```
print("Hello world!\nThis is a\tcode\tcell.")
```

```
Hello world!
This is a	code	cell.
```

You can tell what the type of a cell is by selecting the cell, and looking at the toolbar at the top of the page. For example, try clicking on this cell. You should see the cell type menu displaying "Markdown", like this:



1.4 Command mode and edit mode

When you interact with the notebook, there are two modes:

- **edit mode**
- **command mode**

By default, the notebook starts in *command mode*. To edit a cell, you need to be in *edit mode*.

When you are in command mode, you can press **enter** to switch to edit mode.

- The outline of the cell you currently have selected will turn green, and a cursor will appear.

When you are in edit mode, you can press **escape** to switch to command mode.

- The outline of the cell you currently have selected will turn blue (by default), and the cursor will disappear.

1.4.1 Markdown cells

For example, a markdown cell might look like this in **command mode** (Note: the following few cells are not actually cells – they are images and just look like cells! This is for demonstration purposes only.)

This is a markdown cell.

Then, when you press enter, it will change to **edit mode**:

```
This is a markdown cell.
```

Now, when we press escape, it will change back to **command mode**:

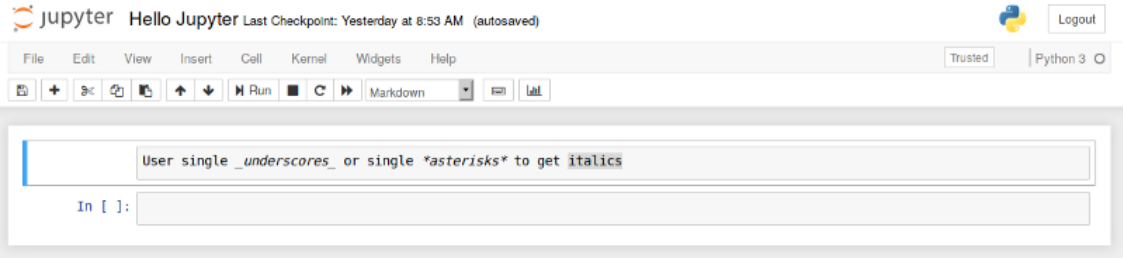
```
This is a markdown cell.
```

However, you'll notice that the cell no longer looks like it did originally. This is because IPython will only *render* the markdown when you tell it to. To do this, we need to "run" the cell by pressing `Ctrl-Enter`, and then it will go back to looking like it did originally:

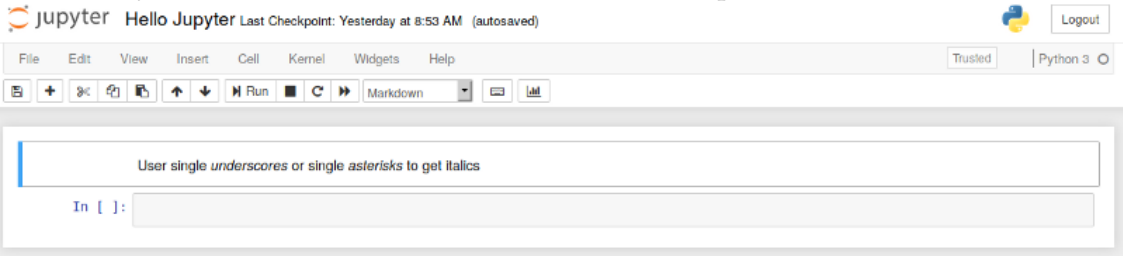
This is a markdown cell.

You will use Markdown cells to give answers to homework questions, and for your final project/paper for this class, so let's learn a little bit of the Markdown syntax.

Italics and boldface Set a new cell to Markdown and then add the following text to the cell:

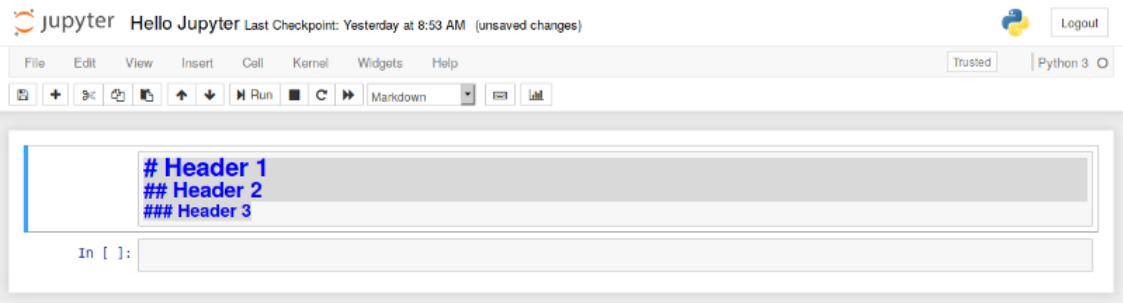


When you run the cell, the output should look like this:

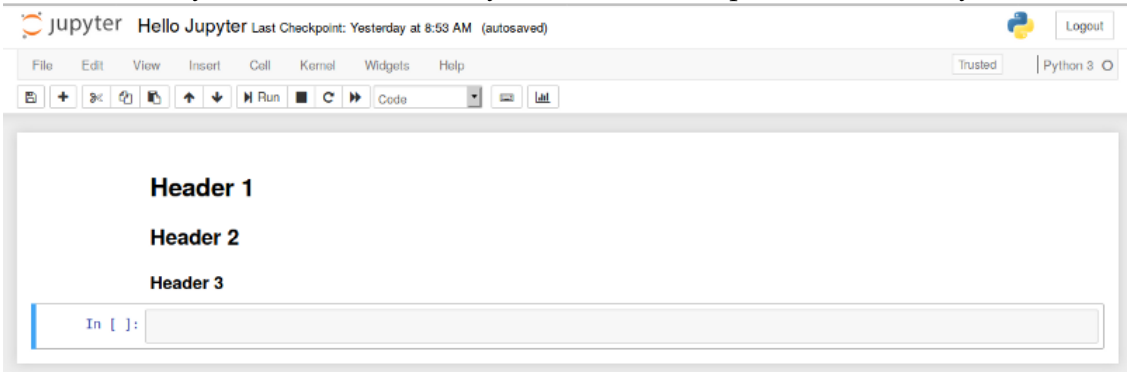


If you would prefer to bold your text, use a double underscore or double asterisk.

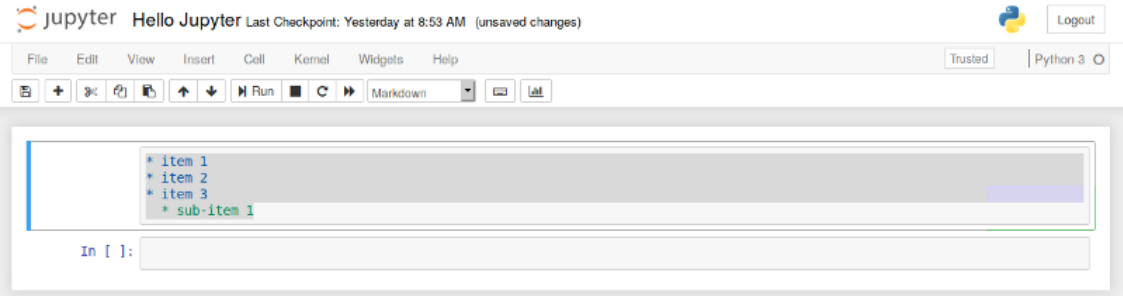
Headers Creating headers in Markdown is also quite simple. You just have to use the humble pound sign. The more pound signs you use, the smaller the header. Jupyter Notebook even kind of previews it for you:



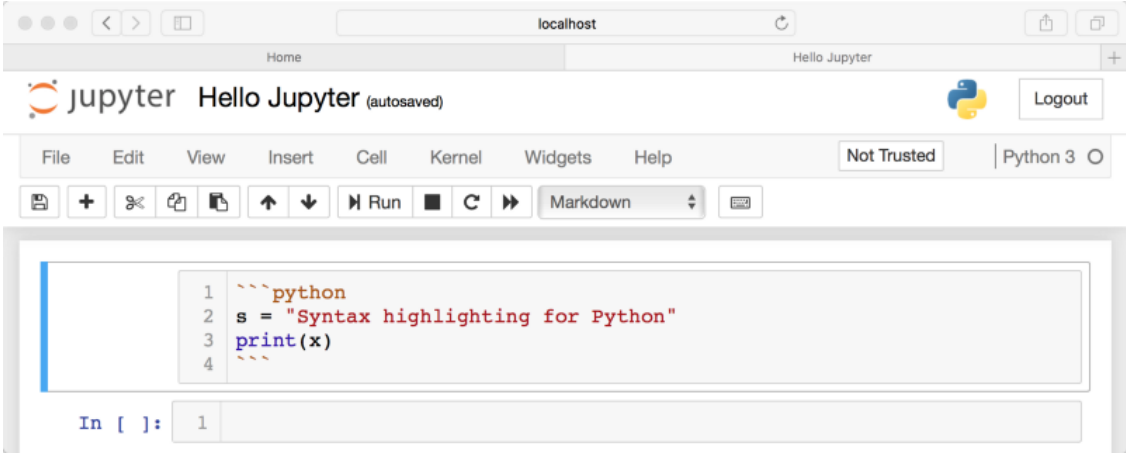
Then when you run the cell, you will end up with a nicely formatted header:



Creating Lists You can create a list (bullet points) by using dashes, plus signs, or asterisks. Here is an example:



Code and Syntax Highlighting If you want to insert a code example that you don't want your end user to actually run, you can use Markdown to insert it. For inline code highlighting, just surround the code with backticks. If you want to insert a block of code, you can use triple backticks and also specify the programming language:



1.4.2 Code cells

For code cells, it is pretty much the same thing. This is what a code cell looks like in command mode (again, the next few cells LOOK like cells, but are just images):

- A code cell in command mode

```
In [ ]: print("this is a code cell")
```

- If we press enter, it will change to **edit mode**:

```
In [ ]: print("this is a code cell")
```

- Pressing escape will make the cell go back to **command mode**:

```
In [ ]: print("this is a code cell")
```

- If we were to press `Ctrl-Enter` like we did for the markdown cell, this would actually *run* the code in the code cell:

```
In [1]: print("this is a code cell")
        this is a code cell
```

1.5 Executing cells

Code cells can contain any valid Python code in them. When you run the cell, the code is executed and the outputs (if any) are displayed.

- You can execute cells with `Ctrl-Enter`, which will keep the cell selected
- You can also execute cells with `Shift-Enter`, which will select the next cell

Try running the following cell and see what it prints out:

[6]:
```
print("Printing cumulative sum from 1-10:")
total = 0
for i in range(1, 11):
    total += i
    print("Sum of 1 to " + str(i) + " is: " + str(total))
print("Done printing numbers.")
```

```
Printing cumulative sum from 1-10:
Sum of 1 to 1 is: 1
Sum of 1 to 2 is: 3
Sum of 1 to 3 is: 6
Sum of 1 to 4 is: 10
Sum of 1 to 5 is: 15
Sum of 1 to 6 is: 21
Sum of 1 to 7 is: 28
Sum of 1 to 8 is: 36
Sum of 1 to 9 is: 45
Sum of 1 to 10 is: 55
Done printing numbers.
```

You'll notice that the output beneath the cell corresponds to the `print` statements in the code.

Here is another example, which only prints out the final sum:

[12]:
```
total = 0
for i in range(1, 11):
    total += i
print(f'The total is: {total}')
```

```
The total is: 55
```

Another way to print something out is to have that thing be the last line in the cell. For example, we could rewrite our example above to be:

[8]:
```
total = 0
for i in range(1, 11):
    total += i
total
```

[8]:
```
55
```

If you want to suppress the automatic printing of the output (if any) resulting from the last line of code in the cell, add a semicolon after it:

[14]:
```
total = 0
for i in range(1, 11):
    total += i
total;
```

Note that only the output (if any) of the last line is automatically displayed. For example, if we wanted to print the total sum and then a message after that, the code below will not do what we want:

[9]:
```
total = 0
for i in range(1, 11):
    total += i
total
print("Done computing total.")
```

```
Done computing total.
```

If you want to print the total, you have to insert an explicit `print` command:

[11]:
```
total = 0
for i in range(1, 11):
    total += i
print(total)
print("Done computing total.")
```

```
55
Done computing total.
```

If you are accustomed to Python2, note that the parentheses are obligatory for the `print` function in Python3.

1.6 The IPython kernel

When you first start a notebook, you are also starting its associated Python3 *kernel*.

- Technically speaking, it is not a stock Python3 kernel, it is an IPython kernel, which is an enhanced Python kernel with facilities for interactive scientific computing.

Whenever you run a code cell, you are telling the kernel to execute the code that is in the cell, and send back to the notebook the output (if any).

Just like if you were typing code at the Python interpreter, you need to make sure your variables are declared before you can use them. What will happen when you run the following cell? Try it and see:

[15]:
```
a
```

```
NameError                                 Traceback (most recent call last)

<ipython-input-15-3f786850e387> in <module>
----> 1 a


NameError: name 'a' is not defined
```

The issue is that the variable `a` does not exist. Modify the cell above so that `a` is declared first:

- for example, you could set the value of `a` to 42 like so: `a = 42`; insert this line of code before you ask what the value of `a` is

Once you have modified the above cell, you should be able to run the following cell (if you haven't modified the above cell, you'll get the same error!):

[17]:
```
print("The value of 'a' is: " + str(a))
```

```
The value of 'a' is: 42
```

Running the above cell should work, because `a` has now been declared. To see what variables have been declared, you can use the `%whos` command:

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

```
Variable        Type        Data/Info
-------------------------------------
a               int         42
i               int         10
json            module      <module 'json' from
'/usr<...>hon3.8/json/__init__.py'>
total           int         55
yapf_reformat   function    <function yapf_reformat at 0x7f1868192040>
```

If you ran the summing examples from the previous section, you'll notice that `total` and `i` are listed under the `%whos` command. That is because when we ran the code for those examples, they also modified the kernel state.

Note that commands beginning with a percent (%) or double percent (%%) are special IPython commands called *magics*. They will **only** work in IPython.

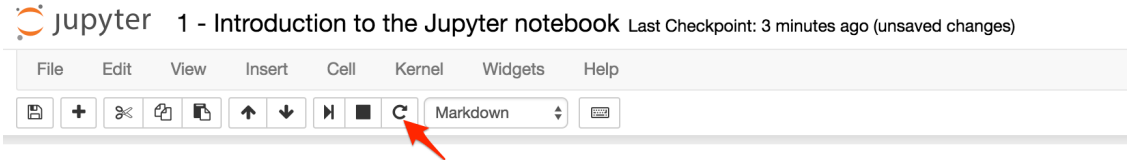
Similarly, commands starting with an exclamation mark (!) are commands that can be run on the background operating system (in the system shell/terminal), and they will only work in IPython.

1.6.1 Restarting the kernel

It is generally a good idea to periodically restart the kernel and start fresh, because you may be using some variables that you declared at some point, but at a later point deleted or updated that declaration.

Your code should ALWAYS be able to work if you run every cell in the notebook, in order, starting from a new kernel.

Again, you can check this in one go by selecting **Restart & Run All** from the **Kernel** menu above. Alternatively, you can first restart the kernel by clicking the restart button:



Then, run all cells in the notebook in order by choosing **Cell→Run All** from the menu above.

There are many keyboard shortcuts for the notebook. To see a full list of these, go to **Help→Keyboard Shortcuts**.

To take the IPython Notebook user interface tour, go to **Help→User Interface Tour**.

Importantly, **Google Colab notebooks are not 'classic' Jupyter notebooks**. They provide most of the same facilities, but:

- the keyboard shortcuts might be different
- some facilities might be missing
- some parts of the interface might be different, e.g., the colors for Command/Edit modes, the way code cells in Command/Edit mode are displayed, etc.

But these are minor differences. Most of the time, you can save a Colab notebook as a Jupyter (`.ipynb`) notebook and run it on a different Jupyter & Python installation without issues, and vice-versa.

And if you run into problems – related to Jupyter, Colab, or Python/Ipython – the web is your friend. A little effort searching around will probably give you a quick solution – or let you know there isn't one, so you can move on (or solve the problem yourself for everyone's benefit).

1.7 Markdown cheatsheet

Here's a Markdown reference website: `https://help.github.com/articles/markdown-basics`

Recall:

- after editing the Markdown, you will need to run the cell so that the formatting appears.
- try selecting *this* cell so you can see what the Markdown looks like when you're editing it
- remember to run the cell again to see what it looks like when it is formatted.

1.8 Additional Resources

- What is Jupyter Notebook? (youtube)
- Jupyter Notebook: An Introduction (Mike Driscoll, RealPython.com)
- Project Jupyter homepage
- List of other programming languages/kernels you can use with Jupyter

[ ]: