

Python for Neuroscientists

Pre-Class Material

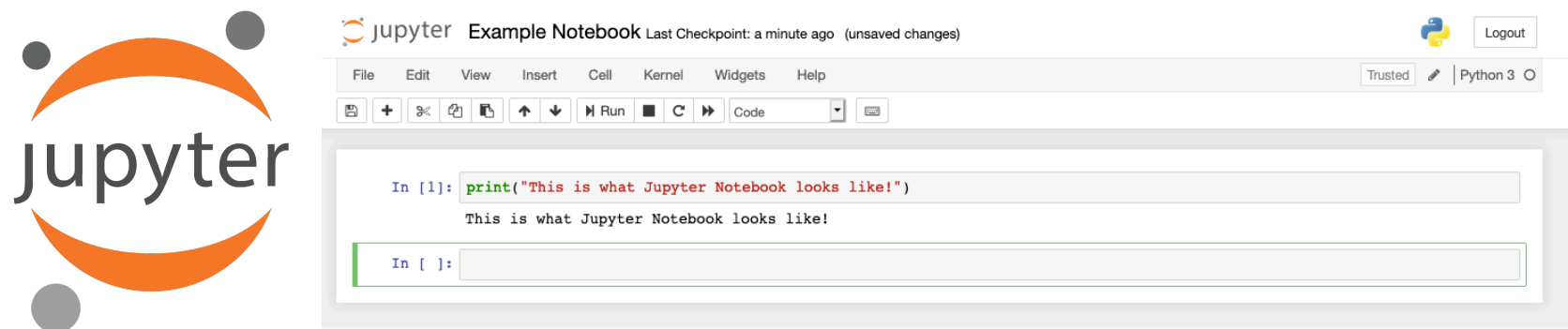
Hi all! The following slides contain instructions and an attempt at providing some intuition for executing basic Python commands in Jupyter Notebook. If you already feel comfortable hosting a Notebook off of your personal computer and importing packages (i.e. numpy, matplotlib), you can stop here :)

Installing Python

If you have not installed Python on your computer, please do so before the first class. We recommend using the Anaconda distribution.



In addition to containing many of the Python libraries you will need in this course (and in life), Anaconda conveniently installs the **Jupyter Notebook** application. Jupyter Notebook is a user-friendly programming environment that will be heavily used in this course.



Please follow the install guide (Python 3.7) below:

<https://docs.anaconda.com/anaconda/install/>

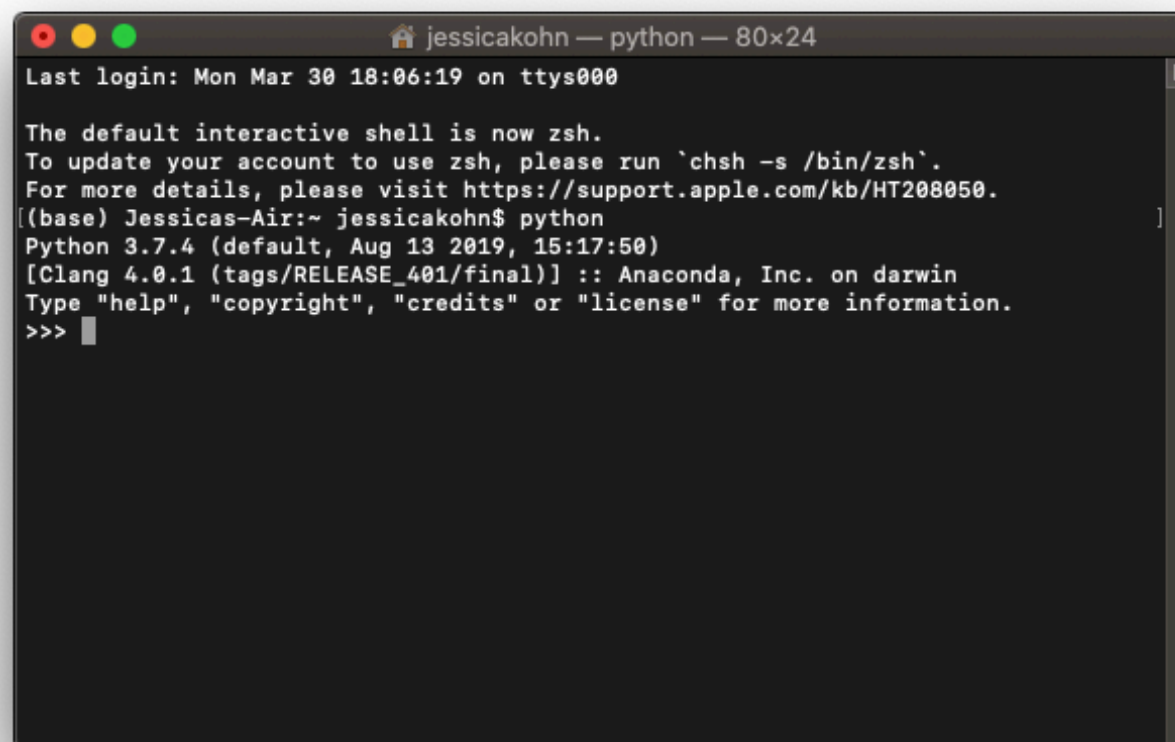
Using Python: Testing Basic Tools

Once Anaconda is installed on your computer, you're ready to go!

By installing this Python distribution, what you've done is give your computer the rules it needs to **interpret commands written in Python into machine language (binary)**. As long as your computer knows that you're speaking to it in Python, it will use these rules to carry out the task that you are asking it to perform.

Let's start with some of the most basic functionality available in Python: simple mathematical tools.

Open Terminal (for Macs - use conda prompt in Windows) and type "python". It should look like this:



```
jessicakohn — python — 80x24
Last login: Mon Mar 30 18:06:19 on ttys000

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
(base) Jessicas-Air:~ jessicakohn$ python
Python 3.7.4 (default, Aug 13 2019, 15:17:50)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Using Python: Testing Basic Tools

What can Python do all on its own?

One example (of many): Python comes with a number of built-in **arithmetic operators**:

+	Addition
-	Subtraction
*	Multiplication
**	Exponential
/	Division
//	Floor Division
%	Modulus

Using the Python shell you opened in the previous slide, try a few of these out. What happens when you type in `1 + 1`? `100/10`? `100//11`?

When you're finished, you can exit the Python shell by typing `exit()`

Using Python: Programming Environments

In the last slide, we learned that using Python is as easy as typing “Python” into your command line interface (CLI - Terminal or conda prompt).

More advanced programming techniques can certainly be used in the shell; however, wouldn't you rather use a gorgeous, intuitive user-interface?

If you use MATLAB, you already understand the benefits of having a programming environment that helps you visualize your code. In contrast to the MATLAB environment that you might be familiar with, Python code can be written in *hundreds* of environments: all you need to do is choose the one that's most appropriate for what you're trying to do.

Using Python: Text Editors

For example, long scripts (like those used to run an application or run in-depth analysis) are often written using **text editors**. Common editors that help with Python formatting and color-coding include Sublime, Atom, and Vim (Vim is often pre-installed). We'll get to these eventually; however, text editors can be a little unwieldy if all you want to do is test small snippets of code. What are our alternatives?



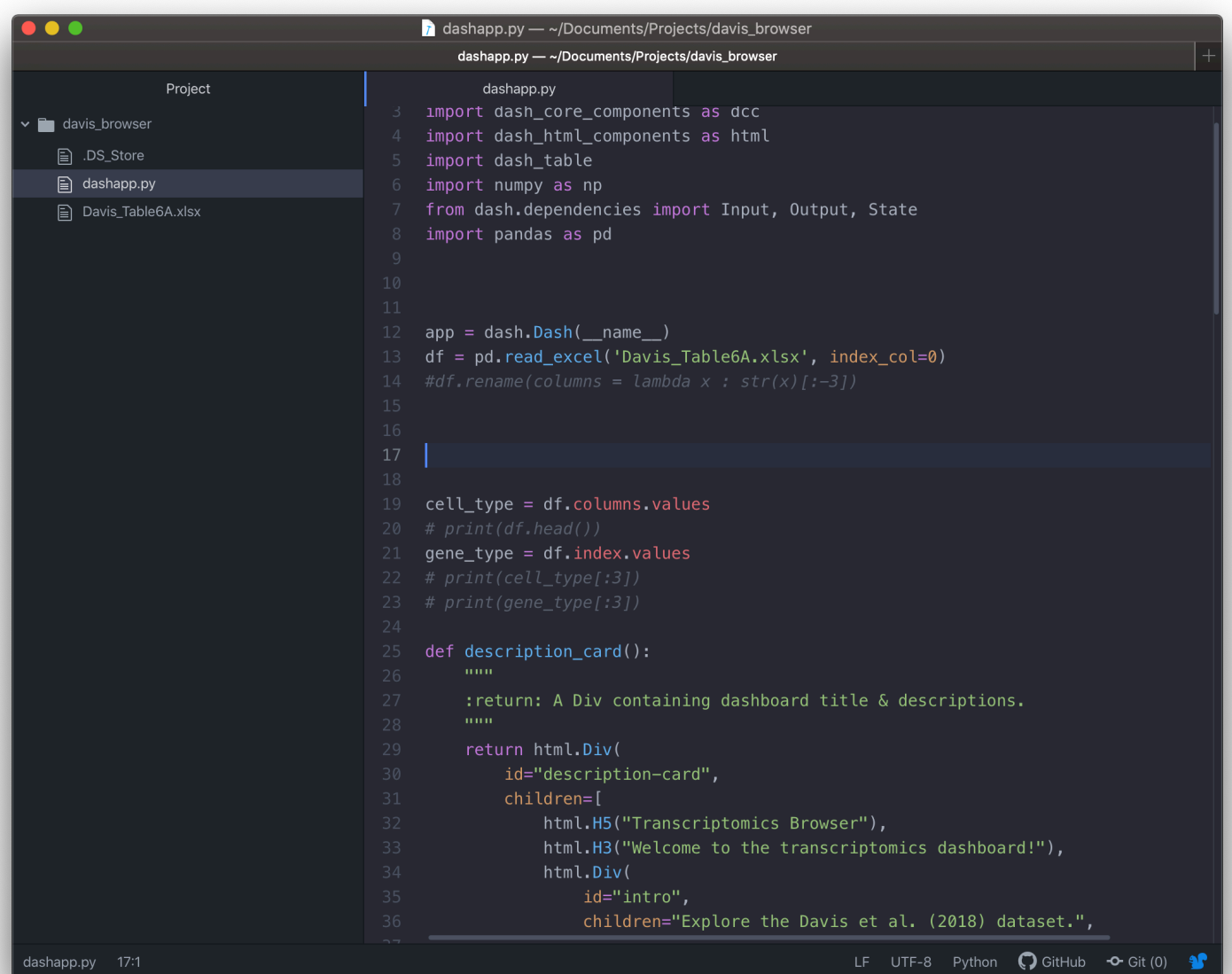
```
import dash
import dash_core_components as dcc
import dash_html_components as html
import dash_table
import numpy as np
from dash.dependencies import Input, Output, State
import pandas as pd

app = dash.Dash(__name__)
df = pd.read_excel('Davis_Table6A.xlsx', index_col=0)
#df.rename(columns = lambda x : str(x)[:3])

cell_type = df.columns.values
# print(df.head())
gene_type = df.index.values
# print(cell_type[:3])
# print(gene_type[:3])

def description_card():
    """
    :return: A Div containing dashboard title & descriptions.
    """
    return html.Div(
        id="description-card",
        children=[
            html.H5("Transcriptomics Browser"),
            html.H3("Welcome to the transcriptomics dashboard!"),
            html.Div(
                id="intro",
                children="Explore the Davis et al. (2018) dataset.",
            ),
        ],
    )

def generate_control_card():
    """
    :return: A Div containing controls for the heatmap.
    """
    return html.Div(
        id="control-card",
        children=[
            html.P("Select cell type(s)"),
            dcc.Dropdown(
```



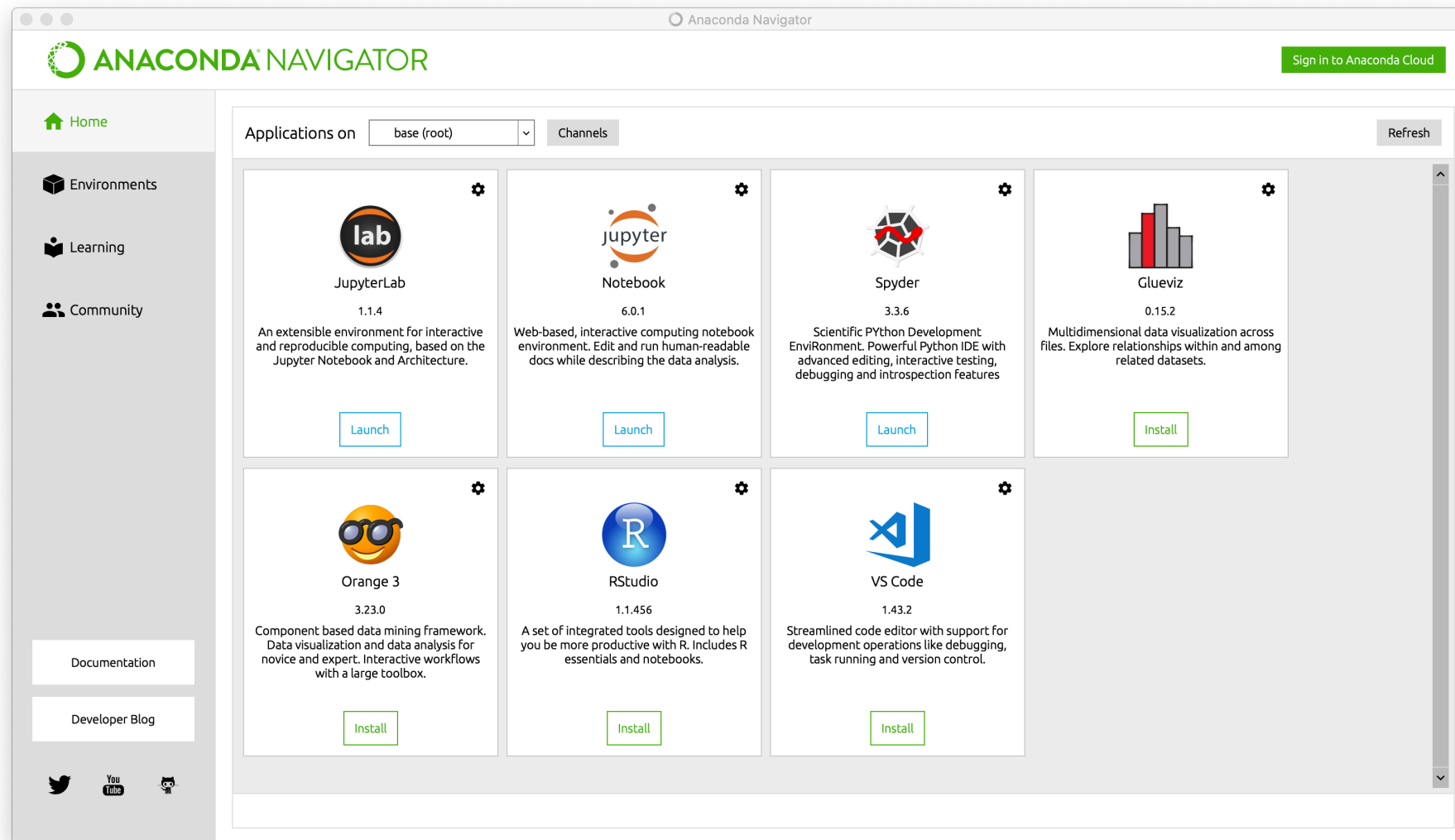
```
dashapp.py — ~/Documents/Projects/davis_browser
dashapp.py — ~/Documents/Projects/davis_browser

3 import dash_core_components as dcc
4 import dash_html_components as html
5 import dash_table
6 import numpy as np
7 from dash.dependencies import Input, Output, State
8 import pandas as pd
9
10
11
12 app = dash.Dash(__name__)
13 df = pd.read_excel('Davis_Table6A.xlsx', index_col=0)
14 #df.rename(columns = lambda x : str(x)[:3])
15
16
17
18
19 cell_type = df.columns.values
20 # print(df.head())
21 gene_type = df.index.values
22 # print(cell_type[:3])
23 # print(gene_type[:3])
24
25 def description_card():
26     """
27     :return: A Div containing dashboard title & descriptions.
28     """
29     return html.Div(
30         id="description-card",
31         children=[
32             html.H5("Transcriptomics Browser"),
33             html.H3("Welcome to the transcriptomics dashboard!"),
34             html.Div(
35                 id="intro",
36                 children="Explore the Davis et al. (2018) dataset.",
37             ),
38         ],
39     )
40
41 def generate_control_card():
42     """
43     :return: A Div containing controls for the heatmap.
44     """
45     return html.Div(
46         id="control-card",
47         children=[
48             html.P("Select cell type(s)"),
49             dcc.Dropdown(
```

The same code in a basic text editor (left) and in Atom (right, with Python-specific formatting). Atom auto-indents and uses color-coding to make the code easier to read.

Using Python: Environments built in to Anaconda

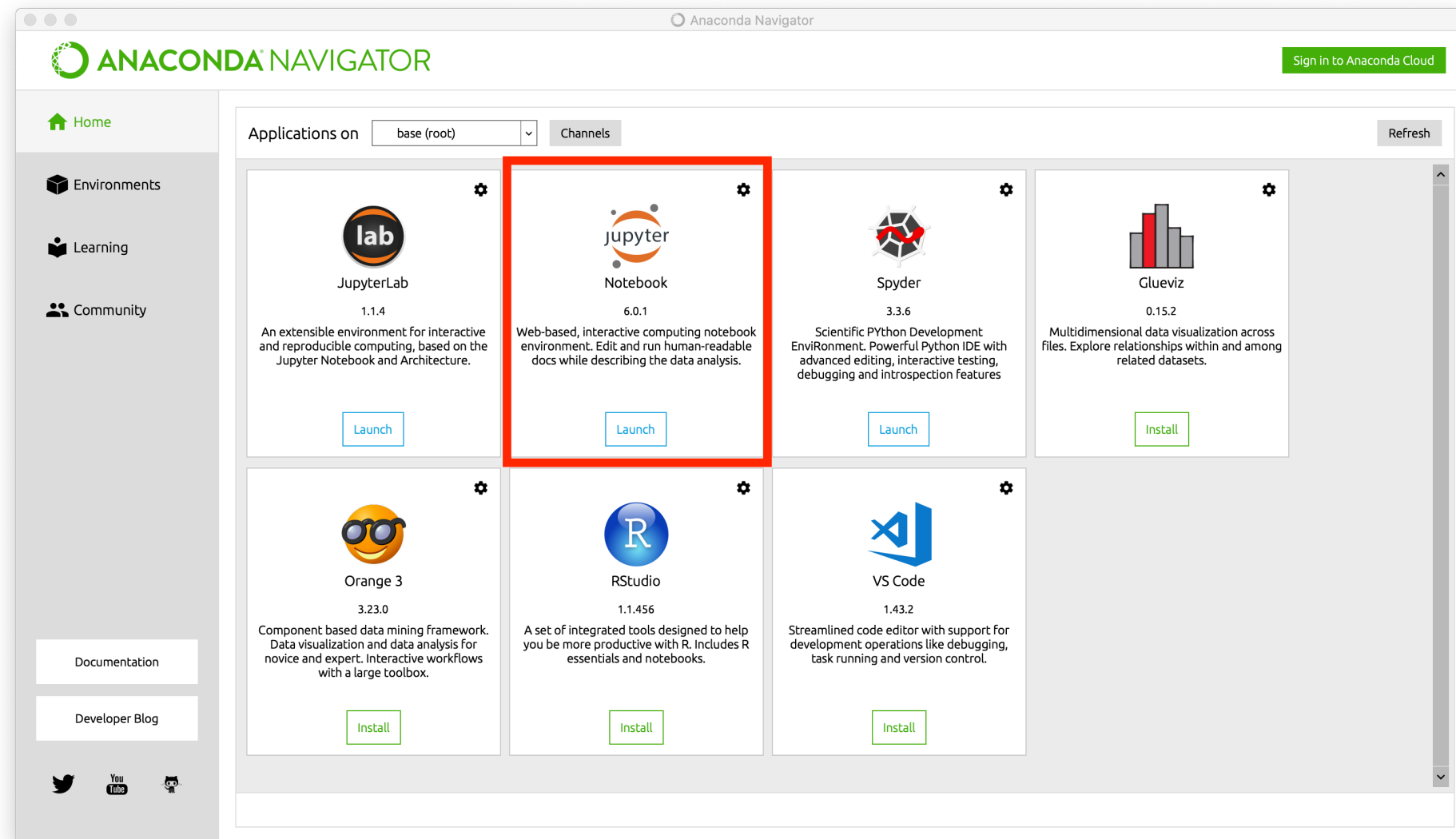
Anaconda comes with a number of applications for using Python. Open Anaconda. The navigator page should look like this:



Instead of the text editors discussed in the previous slide, Anaconda contains environments built by developers for specific tasks. For example, Spyder is a scientific programming environment that functions similarly to MATLAB, while Glueviz and Orange are GUIs for interacting with data.

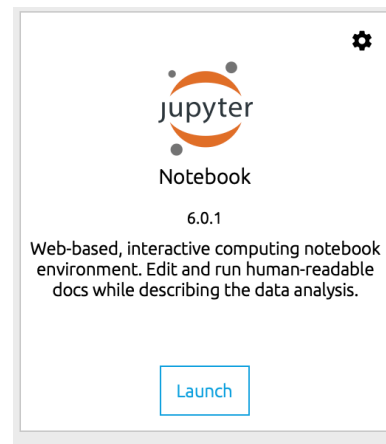
Using Python: Jupyter Notebook

Out of the basic applications available in Anaconda, we'll be focusing on one in particular: Jupyter Notebook.



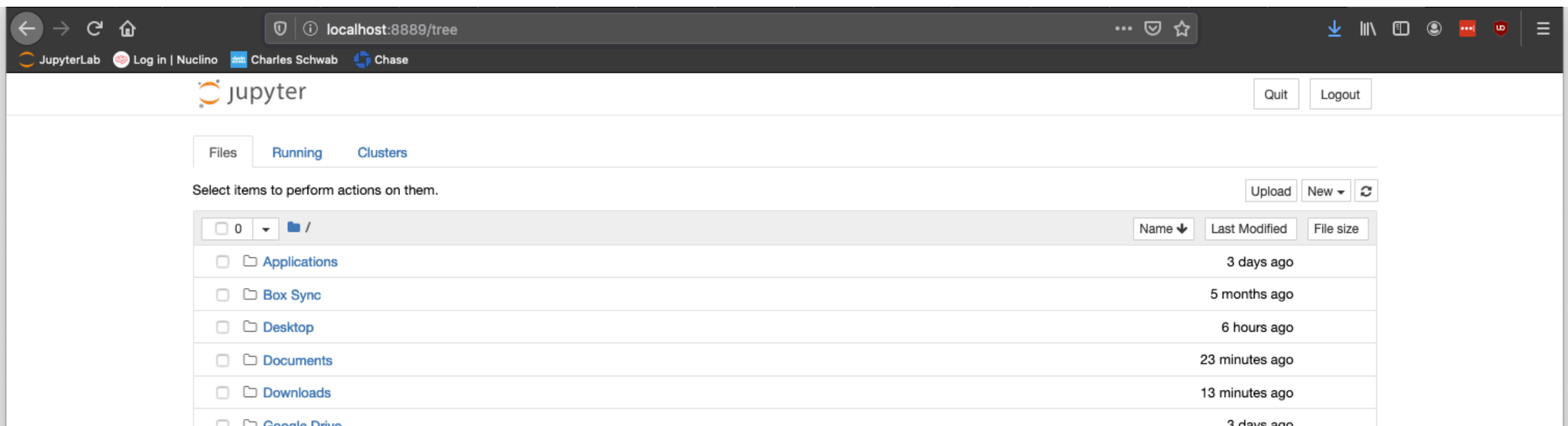
Jupyter Notebook provides a flexible environment for building, visualizing, and testing code. Furthermore, plotting and annotating (via markdown and Latex) in notebooks is incredibly easy, and the .ipynb format of notebooks is easy to share.

Intro to Jupyter Notebook



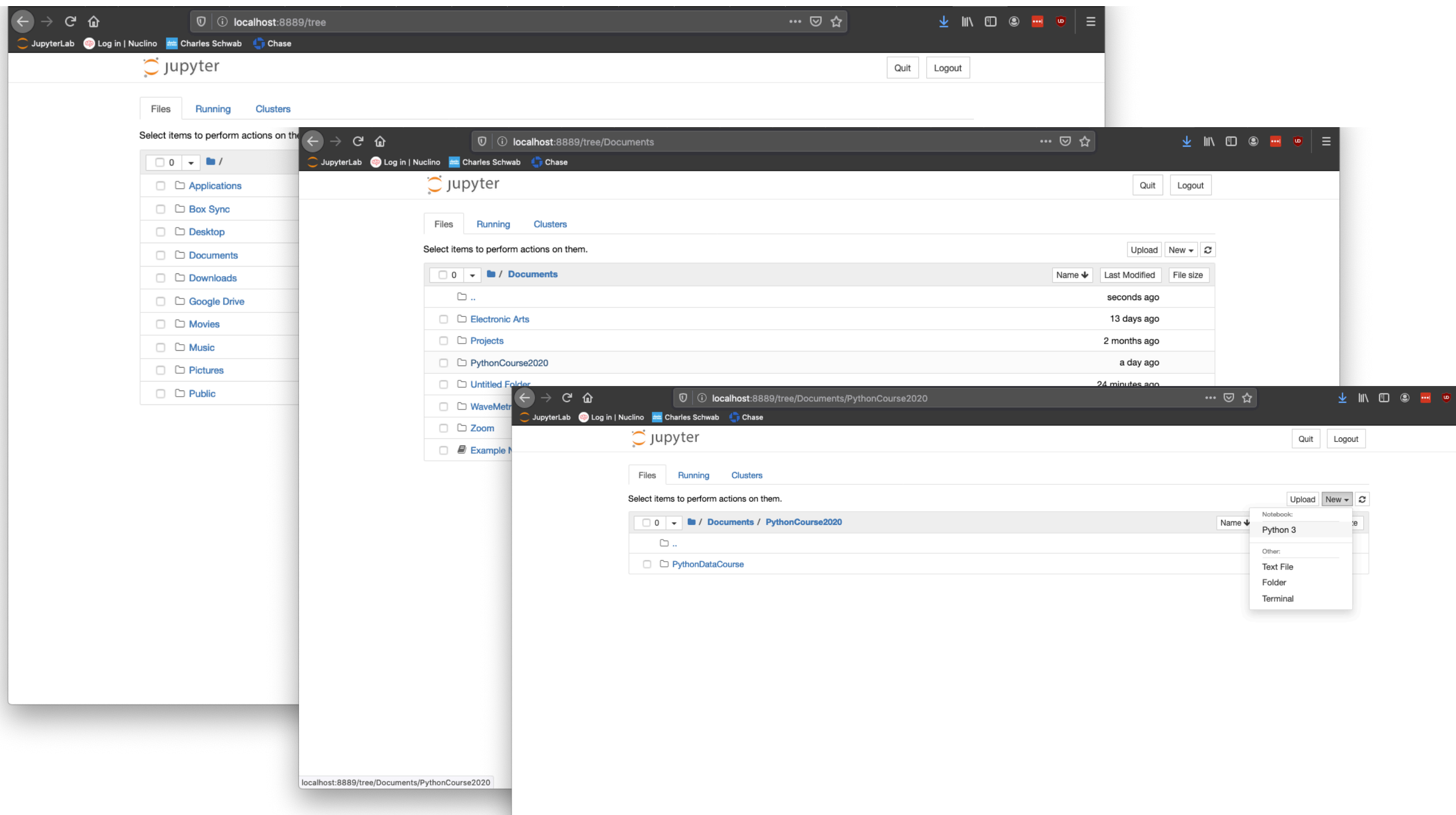
To get started, launch Jupyter Notebook from Anaconda (alternatively, for those who are comfortable, this can be accomplished by navigating to the directory you'd like to open the notebook in via Terminal and running the command “jupyter notebook”).

Off the bat, you might notice that clicking the “Launch” button executes a series of commands in the Terminal, and a new tab opens in your browser that looks like this (depending on the folders you have):



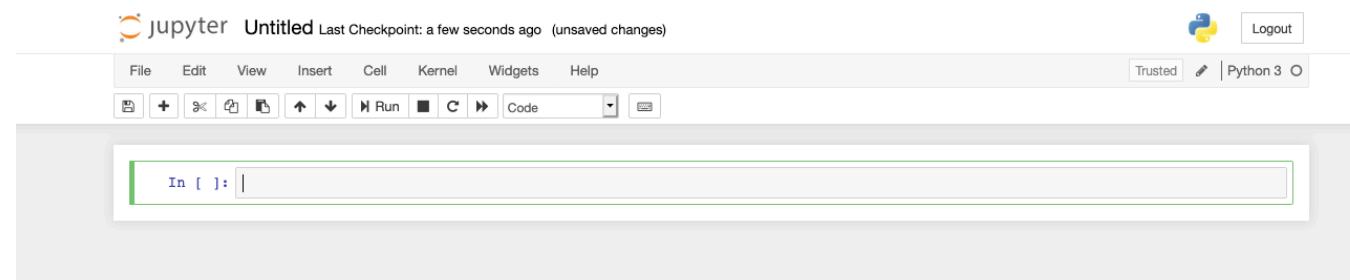
Intro to Jupyter Notebook (cont.)

You can use this interface to navigate to the location in which you'd like to store your work for the remainder of this class (I personally like to create specific folders within my Documents folder, but that's just me). Once in the directory of your choice, click “New”, followed by “Python 3”.

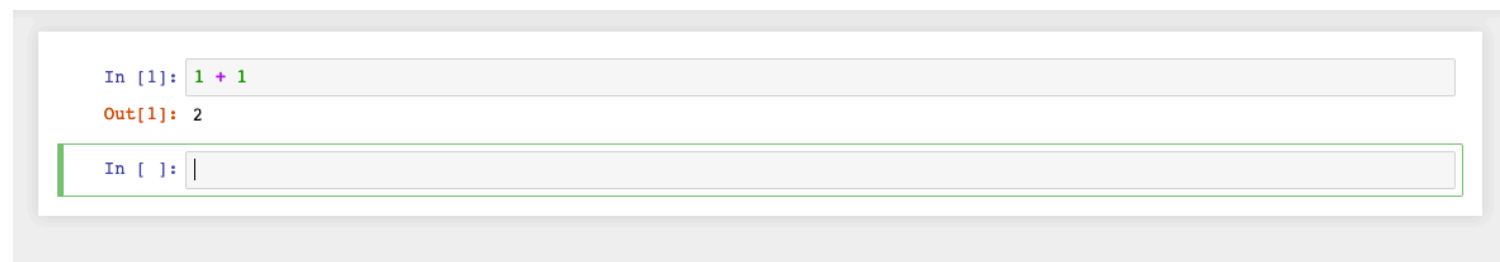


Intro to Jupyter Notebook (cont.)

Awesome! At this point, you should have a fresh new notebook just waiting for you to write some brilliant code. It'll look like this:



Try the same mathematical commands we ran in the Python shell earlier. To run a command in Jupyter Notebook, press “Shift” then “Enter,” or hit the run button.



You'll notice a few things happened when we ran our code: 1.) an “out” window containing our output appeared, and 2.) a new input window opened. This is one of the great strengths of Jupyter Notebook - we can actually run a new line of code in this new input window *without losing sight of what we've already written*. We'll get into this more deeply later on, but give it a try now.



Python Packages/modules

So far, what you've done is execute some basic Python commands in Jupyter Notebook. Sure, it's nice to be able to see what you just did...but you're not going to get very far with a limited set of tools. This is where packages and modules come in.

Before class begins, the last thing we want to make sure that you have a handle on is the concept of a **Python package**.

What is a package? To put it simply, a package (which contains modules) is just a bunch of code in a directory. You can almost think of it as a bunch of folders, organized by the function of the code each folder contains. The majority of these folders were installed on your computer when you installed Anaconda, and contain code for myriad uses. Rather than trying to import EVERY package or module there ever was, which would take ages, Python asks us to specify which ones we're going to use each time we program in a new file.

In contrast to MATLAB, which comes with all of its functions already built-in, Python packages are constantly being edited, expanded, and developed by the general public (don't worry, there are safeguards in place to ensure you can trust the code in any of the major packages we'll be covering in the next few classes).

For a more in-depth explanation, check out this article:

<https://docs.python.org/3/tutorial/modules.html#>

Python Packages (cont.)

Let's try an example...say we want to take the mean of the array [1,2,3,4,5]. Sure, we could say $(1 + 2 + 3 + 4 + 5) / 5$, right? But that's not sustainable if we have more complicated processes we want to execute, or if we have a large array of numbers.

If you're familiar with MATLAB, you'd say "why don't I just use the mean function?" You're on the right track! But give that a try with our example and see what happens.

```
In [1]: mean([1,2,3,4,5])

-----
NameError                                Traceback (most recent call last)
<ipython-input-1-54166c2a3f59> in <module>
----> 1 mean([1,2,3,4,5])

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

As you can see, we get an error: the name 'mean' is not defined- we need to tell our notebook what we're referring to when we say "mean".

To do that, we need to 1.) import a package that contains a **function** to calculate a mean, and 2.) tell our notebook **where** in the package to find that function

In Python, the convention for doing this is:

```
import [PACKAGE NAME] as [ABBREVIATION]
```

Python Packages (cont.)

The first package we'd like you to get to know is called **numpy** (rhymes with “um, pie”... as in “numerical python”... not “bumpy”). numpy is usually abbreviated “np” (to make typing it out faster), and is one of the core packages that Python programmers use for performing computations on arrays. Plan on typing this command into almost every .py or .ipynb file you make:

```
In [1]: import numpy as np

In [ ]: |
```

What we've just done is tell our notebook that we want it to be able to run every piece of code that's available in the numpy package.

But how do we know what numpy can do? This is where the real beauty of Python as an open source language comes in - there's a package for **nearly everything** ([counting sea lions from aerial photos? sure.](#)), **and nearly every package has online documentation.**

Let's try Googling “numpy mean” - it seems like a silly answer to “just Google it,” but trust us when we say that being able to Google search yourself out of a hole is 90% of learning Python.

docs.scipy.org › doc › numpy › reference › generated › numpy.mean... ▼
[numpy.mean — NumPy v1.17 Manual - Numpy and Scipy](#)
Compute the arithmetic mean along the specified axis. Returns the average of the array elements. The average is taken over the flattened array by default, ...

Python Packages (cont.)

```
docs.scipy.org › doc › numpy › reference › generated › numpy.mean...  
numpy.mean — NumPy v1.17 Manual - Numpy and Scipy  
Compute the arithmetic mean along the specified axis. Returns the average of the array  
elements. The average is taken over the flattened array by default, ...  
  
Mean  
mean[] numpy.mean(a,  
axis=None, dtype=None, out=None  
...
```

As you can see, the first result looks pretty promising - it's from a documentation site, and it's meant to show you how to use the code.

numpy.mean

```
numpy.mean(a, axis=None, dtype=None, out=None, keepdims=<no value>) \[source\]  
Compute the arithmetic mean along the specified axis.  
  
Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the  
specified axis. float64 intermediate and return values are used for integer inputs.  
  
Parameters: a : array_like  
Array containing numbers whose mean is desired. If a is not an array, a conversion is attempted.  
  
axis : None or int or tuple of ints, optional  
Axis or axes along which the means are computed. The default is to compute the mean of the flattened  
array.  
New in version 1.7.0.  
If this is a tuple of ints, a mean is performed over multiple axes, instead of a single axis or all the axes as  
before.
```

The documentation explains that numpy's mean function needs to take in an array (a data type you're likely familiar with - but we'll touch on briefly in class) containing the values you want to take the mean of. As you can see by the "optional" in the rest of the parameters, the function doesn't actually need anything else - it just has additional functionality that we will eventually learn more about.

Python Packages (cont.)

Let's give it a shot - the last thing you need to know is that when you're telling your notebook that you want it to use numpy's "mean" function, the convention is to provide the name you **imported the package as** (np), followed by a dot, followed by the function. Like this:

```
In [2]: np.mean([1,2,3,4,5])
```

```
Out[2]: 3.0
```

One last thing: *you can import a package as **anything***. "np" is a convention because standardization is nice and typing "np" is easy, but we could also use:

```
In [4]: import numpy as bananas
```

```
In [5]: bananas.mean([1,2,3,4,5])
```

```
Out[5]: 3.0
```

Not saying you should do this - but do keep in mind that Python is truly free-form. Whatever you've written is the law in your notebook, so always check your import syntax if you hit a bug early on.

Finally, as a side note - we could also import **just** numpy's mean function by typing "from numpy import mean as mean", and then typing "mean([1,2,3,4,5]) to get the same result. We don't recommend doing this for numpy functions - just stick with the np.mean format for now...but thought we'd let you know :) Easy!

Python Package Managers

In the previous few slides, you imported a Python package that's already installed on your computer... but what happens if you're browsing the inter webs and find a really cool new package that you'd like to use? Enter: **package managers!**

A package manager is a utility that simplifies the downloading/installation of Python packages from the internet onto your computer.

The two most common package managers you'll encounter are **pip** and **conda**. Pip ("pip installs packages") is most common; however, [we recommend using conda \(which is released by Anaconda\) in any scenario where it can be used](#) (although some packages will only be able to be installed via pip...in that case don't worry too much).

Python Package Managers (cont.)

Let's say there's a cool package called "antigravity" that you really want to try. First, in your Jupyter Notebook, try importing it...unless you've magically installed it before this course, you're going to get an error!

So how do we install it? Open your terminal, type "pip install antigravity" (normally we'd say use "conda install," but this example isn't able to be installed by conda...sorry! Bad example?), and run the command. It should look like this:

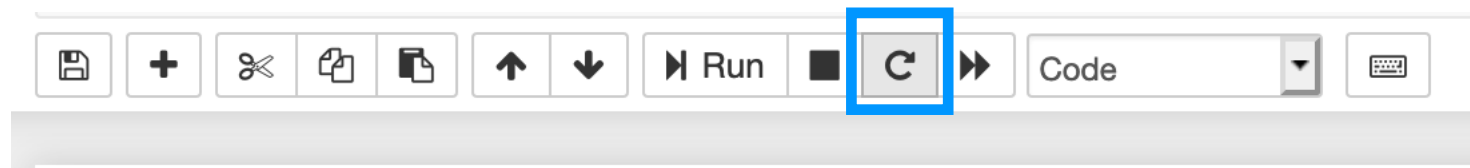
```
(base) Jessicas-Air:documents jessicakohn$ pip install antigravity
Collecting antigravity
  Downloading https://files.pythonhosted.org/packages/07/9c/11a6803be0d87d8d408bd244917ad27556d78185c4b7c4fd535bf06f30a4/antigravity-0.1.zip
Building wheels for collected packages: antigravity
  Building wheel for antigravity (setup.py) ... done
  Created wheel for antigravity: filename=antigravity-0.1-cp37-none-any.whl size=1363 sha256=292e46b70baadb901587a8182520e0b4bbd01f5ba7f9d44ae74e47888e7dd367
  Stored in directory: /Users/jessicakohn/Library/Caches/pip/wheels/cd/f4/eb/b084b0b8a7089faaf881899a3e6130e613ea2909f092e6cd8e
Successfully built antigravity
Installing collected packages: antigravity
Successfully installed antigravity-0.1
```

What pip (or conda, in most scenarios) just did was search online for the package called "antigravity" (no packages registered with pip or conda will share the same name), pull all of its related code, and put it into the Python libraries that are installed on your computer.

Python Package Managers (cont.)

Now, go back to your Jupyter Notebook. Before you'll have access to the new package you've installed in Jupyter, you'll need to restart the kernel (the process on your computer running the notebook - it doesn't know you've installed a new package yet).

To restart the kernel, click this button:



Let's see what happens when you import the package!

Try importing `antigravity` the same way we imported `numpy` earlier.

(this one's for you, Gucky)

Python Package Managers (cont.)

How interesting! It turns out that **the entire purpose** of the `antigravity` package is to open a pro-Python xkcd comic whenever you import it :)

This is a silly example just to get you used to the idea of importing packages that aren't default-installed via Anaconda. Before the first class, we suggest that you do a quick web search for useful Python packages to get an idea of the incredible open-source offerings out there!

We're done for now - see ya soon

At this point, you have more than enough background on how to use Jupyter Notebook for the first class - the expectation will be that you can immediately load a Jupyter Notebook to follow along on your own computer, and are comfortable importing numpy and running commands. If you feel like you're not quite there yet, be sure to send any questions to the Slack channel!

To exit your notebook, simply save and close your browser window - be sure to quit the process in Terminal as well, either by terminating it in the command line or quitting Terminal.

