CS-S109A Introduction to Data Science

Lab 0: Getting Started with Jupyter Notebooks

Harvard University
Summer 2020
Instructor: Kevin Rader
Authors: Rahul Dave, David Sondak, Will Claybaugh, Pavlos Protopapas, Chris Tanner, Eleni Kaxiras, and Kevin Rader

---

In [ ]:
```
## RUN THIS CELL TO GET THE RIGHT FORMATTING
import requests
from IPython.core.display import HTML
styles = requests.get("https://raw.githubusercontent.com/Harvard-IACS/2018-CS109A/master/content/styles/
HTML(styles)
```

In [ ]:
```
PATHTOSOLUTIONS = '../solutions'
```

Programming Expectations

All assignments for this class will use Python and the browser-based iPython notebook format you are currently viewing. Programming at the level of CS 50 is a prerequisite for this course. If you have concerns about this, come speak with any of the instructors.

We will refer to the Python 3 documentation (https://docs.python.org/3/) in this lab and throughout the course.

Learning Goals

This introductory lab is a condensed introduction to Python numerical programming. By the end of this lab, you will feel more comfortable:

- Learn about anconda environments and setup your own with the necessary dependencies
- Writing short Python code to create lists, perform arithmetic operations, and print output

Part 1: Set up a Conda Python Environment and Clone the Class Repository

On Python installation packages

There are two main installing packages for Python, `conda` and `pip`. Pip is the Python Packaging Authority's recommended tool for installing packages from the **Python Package Index (PyPI)**. `Conda` is a cross platform package and environment manager that installs and manages conda packages from the **Anaconda repository** and **Anaconda Cloud**. Conda does not assume any specific configuration in your computer and will install the Python interpreter along with the other Python packages, whereas `pip` assumes that you have installed the Python interpreter in your computer. Given the fact that most operating systems do include Python this is not a problem.

If I could summarize their differences into a sentence it would be that conda has the ability to create **isolated environments** that can contain different versions of Python and/or the packages installed in them. This can be extremely useful when working with data science tools as different tools may contain conflicting requirements which could prevent them all being installed into a single environment. You can have environments with pip but would have to install a tool such as virtualenv or venv. You may use either, we recommend `conda` because in our experience it leads to fewer incompatibilities between packages and thus fewer broken environments.

Conclusion: Use Both. Most often in our data science environments we want to combining pip with conda when one or more packages are only available to install via pip. Although thousands of packages are available in the Anaconda repository, including the most popular data science, machine learning, and AI frameworks but a lot more are available on PyPI. Even if you have your environment installed via `conda` you can use `pip` to install individual packages

(source: anaconda site (https://www.anaconda.com/understanding-conda-and-pip/))

Installing Conda

- First check if you have conda

In **MacOS** or **Linux** open a Terminal window and at the prompt type

```
conda –V
```

If you get the version number (e.g. `conda 4.6.14`) you are all set! If you get an error, that means you do not have Anaconda and would be a good idea to install it.

- If you do not have it, you can install it by following the instructions:

Mac : https://docs.anaconda.com/anaconda/install/mac-os/ (https://docs.anaconda.com/anaconda/install/mac-os/)

Windows : https://docs.anaconda.com/anaconda/install/windows (https://docs.anaconda.com/anaconda/install/windows) (Note: #8 is important: DO NOT add to your path. The reason is that Windows contains paths that may include spaces and that clashes with the way `conda` understands paths.)

- If you do have anaconda consider upgrading it so you get the latest version of the packages:

```
conda update conda
```

Conda allows you to work in 'computing sandboxes' called environments. You may have environments installed on your computer to access different versions of Python and different libraries to avoid conflict between libraries which can cause errors.

What are environments and do I need them?

Environments in Python are like sandboxes that have different versions of Python and/or packages installed in them. You can create, export, list, remove, and update environments. Switching or moving between environments is called activating the environment. When you are done with an environments you may deactivate it.

For this class we want to have a bit more control on the packages that will be installed with the enviromnent so we will create an environment with a so called YAML file called `cs109a.yml` . Originally YAML was said to mean *Yet Another Markup Language* referencing its purpose as a markup language with the yet another construct, but it was then repurposed as *YAML Ain't Markup Language* [source:wikipedia]. This is included in the Lab directory in the class git repository.

Creating an environment from an environment.yml file

Using your browser, visit the class git repository https://github.com/orgs/Harvard-IACS/s109a-2020 (https://github.com/orgs/Harvard-IACS/s109a-2020)

Go to `content --> labs/ --> lab1` and look for the cs109a.yml file. Download it to a local directory in your computer.

Then in the Terminal again type

```
conda env create -f {PATH-TO-FILE}/cs109a.yml
```

Activate the new environment:

```
source activate cs109a
```

You should see the name of the environment at the start of your command prompth in parenthesis.

Verify that the new environment was installed correctly:

```
conda list
```

This will give you a list of the packages installed in this environment.

References

Manage conda environments (https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html)

Clone the class repository

In the Terminal type:

```
git clone https://github.com/orgs/Harvard-IACS/s109a-2020.git
```

Starting the Jupyter Notebook

Once all is installed go in the Terminal and type

```
jupyter notebook
```

to start the jupyter notebook server. This will spawn a process that will be running in the Terminal window until you are done working with the notebook. In that case press `control-C` to stop it.

Starting the notebook will bring up a browser window with your file structure. Look for the s109a-2020 folder. It should be where you cloned it previously. When you visit this folder in the future, and while in the top folder of it, type

```
git pull
```

This will update the contents of the folder with whatever is new. Make sure you are at the top part of the folder by typing

```
pwd
```

which should give you `/s109A-2020/`

For more on using the Notebook see: https://jupyter-notebook.readthedocs.io/en/latest/ (https://jupyter-notebook.readthedocs.io/en/latest/)

Part 2: Getting Started with Python

Importing modules

All notebooks should begin with code that imports *modules*, collections of built-in, commonly-used Python functions. Below we import the Numpy module, a fast numerical programming library for scientific computing. Future labs will require additional modules, which we'll import with the same syntax.

```
import MODULE_NAME as MODULE_NICKNAME
```

In [ ]:
```
#import numpy as np #imports a fast numerical programming library
```

Now that Numpy has been imported, we can access some useful functions. For example, we can use `mean` to calculate the mean of a set of numbers.

In [ ]:
```
my_list = [1.2, 2, 3.3]
print(my_list)
```

Calculations and variables

In [ ]:
```
# // is integer division
1/2, 1//2, 1.0/2, 3*3.2
```

The last line in a cell is returned as the output value, as above. For cells with multiple lines of results, we can display results using `print`, as can be seen below.

In [ ]:
```
print(1 + 3.0, "\n", 9, 7)
5/3
```

We can store integer or floating point values as variables. The other basic Python data types -- booleans, strings, lists -- can also be

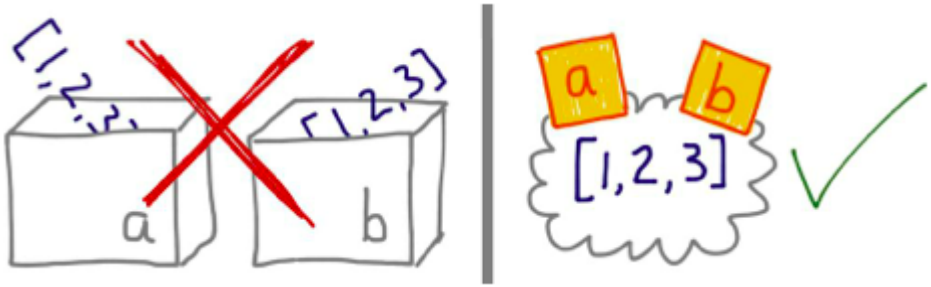
stored as variables.

In [ ]:
```
a = 1
b = 2.0
```

Here is the storing of a list

In [ ]:
```
a = [1, 2, 3]
```

Think of a variable as a label for a value, not a box in which you put the value



(image: Fluent Python by Luciano Ramalho)

In [ ]:
```
b = a
b
```

This DOES NOT create a new copy of `a`. It merely puts a new label on the memory at a, as can be seen by the following code:

In [ ]:
```
print("a", a)
print("b", b)
a[1] = 7
print("a after change", a)
print("b after change", b)
```

Tuples

Multiple items on one line in the interface are returned as a *tuple*, an immutable sequence of Python objects. See the end of this notebook for an interesting use of `tuples`.

In [ ]:
```
a = 1
b = 2.0
a + a, a - b, b * b, 10*a
```

type()

We can obtain the type of a variable, and use boolean comparisons to test these types. VERY USEFUL when things go wrong and you cannot understand why this method does not work on a specific variable!

In [ ]:
```
type(a) == float
```

In [ ]:
```
type(a) == int
```

In [ ]:
```
type(a)
```

For reference, below are common arithmetic and comparison operations.

Operator	Description	Example
+	adds values on either side	1.2 + 2 = 3.2
-	subtracts the right value from the left	1.2 - 0.2 = 1.0
*	multiplies values on either side	1.2 * 2 = 2.4
/	divides the left value by the right	4 / 2 = 2.0
%	divides the left value by the right and returns the remainder	4 % 3 = 1
**	exponentiate the left value by the right	3**2 = 9
//	divides the left value by the right and removes the decimal part	3//2 = 1

Operator	Description	Example
==	checks if values on either side are equal	1 == 2 is False
!=	checks if values on either side are unequal	1 != 2 is True
>	checks if left value is greater	1 > 2 is False
<	checks if left value is smaller	1 < 2 is True
>=	checks if left value is greater or equal	2 >= 2 is True
<=	checks if left value is smaller or equal	1 <= 2 is True

EXERCISE 1: Create a tuple called `tup` with the following seven objects:

- The first element is an integer of your choice
- The second element is a float of your choice
- The third element is the sum of the first two elements
- The fourth element is the difference of the first two elements
- The fifth element is the first element divided by the second element
- Display the output of `tup` . What is the type of the variable `tup` ? What happens if you try and chage an item in the tuple?

In [ ]:
```
# your code here
```

In [ ]:
```
# TO RUN THE SOLUTIONS
# 1. uncomment the first line of the cell below so you have just %load
# 2. Run the cell AGAIN to execute the python code, it will not run when you execute the %load command!
```

In [ ]:
```
# %load ../solutions/exercise1.py
```