

Fundamentals of Information Systems

Python Programming (for Data Science)

Master's Degree in Data Science

Gabriele Tolomei

gtolomei@math.unipd.it

University of Padua, Italy

2017/2018

October 11, 2018

Lecture 1: Introduction and Environment Setup

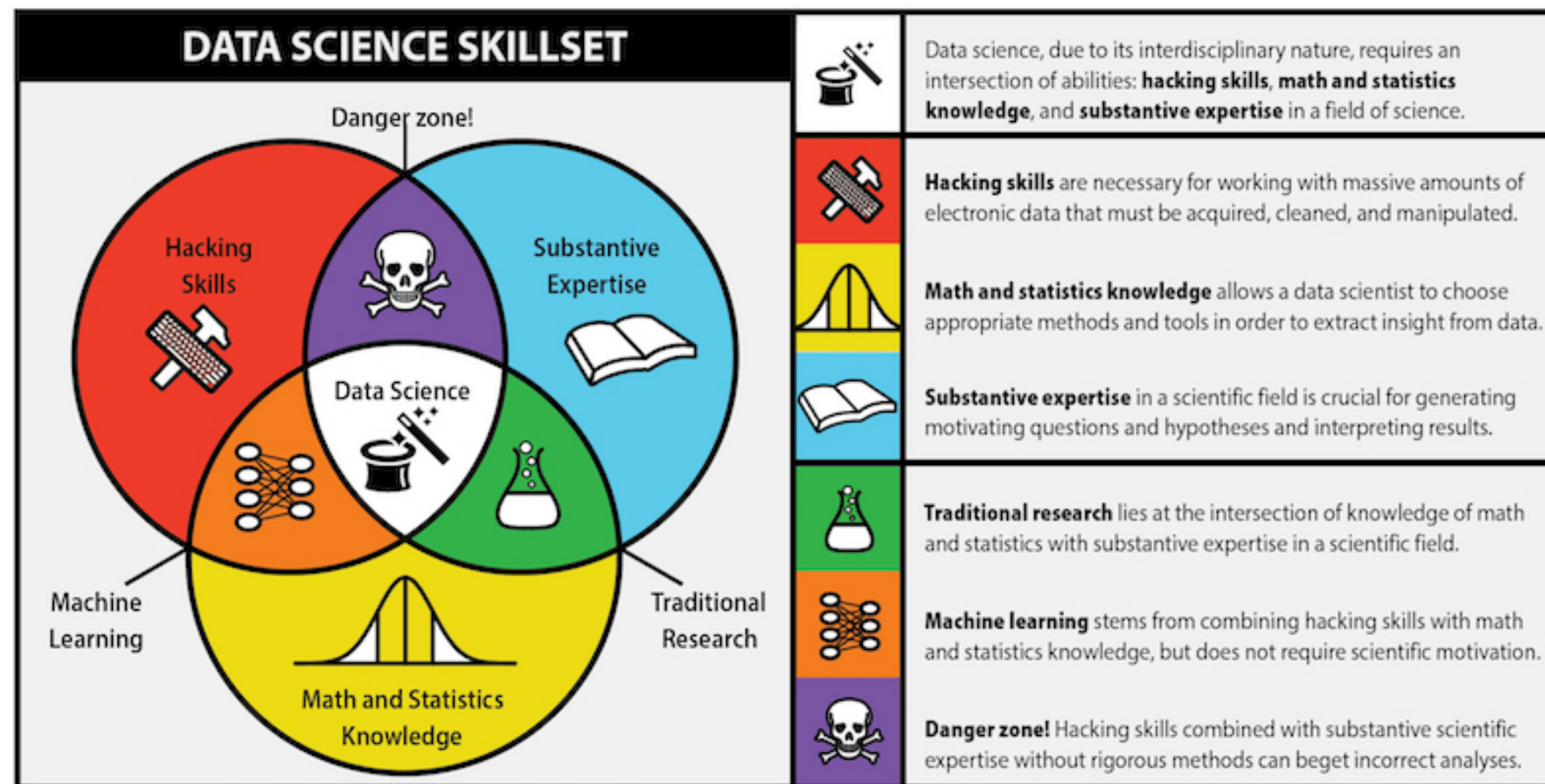
What is Data Science?

According to [Wikipedia](#):

"Data Science is an **interdisciplinary** field about scientific methods, processes, and systems to extract **knowledge** or insights from **data** in various forms, either structured or unstructured."

What is Data Science?

A picture is worth a thousand words...



What kind of Data?

Mostly, **structured data**, which is a (vague) term that encompasses many different common forms of data, such as:

- Tabular data like those stored in RDBMS or tab- or comma-separated text files, each column of a different type (*string, numeric, date*, etc.).
- Multidimensional arrays (matrices).
- Multiple tables of data interrelated by key columns (what would be primary or foreign keys for a SQL user).
- Evenly or unevenly spaced time series.
- In general, any source of data with a *schema* attached.

What kind of Data?

- The list above is by no means complete!
- Often, *unstructured* data sets can be transformed into a structured form that is more suitable for analysis and modeling.
- If not, it may be possible to **extract features** from a data set into a structured form. For example:
 - A collection *text documents* could be processed into a word frequency table which could then be used to perform sentiment analysis.
 - A set of *images* can be transformed into RGB matrices of fixed size to perform some visual recognition task (cat vs. dog).

Why Python?

- Since its first appearance in 1991, *Python* has become one of the most popular programming languages, along with *Perl*, *Ruby*, and others.
- It is often referred to as a *scripting language* to emphasize that it can be used to *quickly* write small programs (or scripts).
- Such "rapid prototyping" ability is one of the difference between Python and "established" programming languages like *C/C++* or *Java*.
- However, the term "scripting language" might be misleading as it seems that Python cannot be used for building more complex software systems.

Why Python for Data Science?

- Alternatives to Python in the data analysis field are still widely used, e.g., R, MATLAB, SAS, Stata, etc.
- In the last 10 years, Python has become one of the most important languages for **data science**, **machine learning** both in academia and industry.
 - Highly active and collaborative community;
 - Huge library support (e.g., **numpy**, **pandas**, **scikit-learn**).
- As of today, Python is the primary language for anyone approaching data science (and, possibly, computer programming in general).

Python as a glue

- Easy to integrate with existing C, C++, and FORTRAN code for scientific computing.
- Modern computing environments share a similar set of legacy FORTRAN and C libraries for doing linear algebra, optimization, integration, fast fourier transforms, and other such algorithms.
- *Portability*: Python code can be executed on any platform (provided there is a Python implementation for that platform)

One language fits (almost) all

- Before Python, organizations used to have 2 separate development stages:
 1. Research, prototyping, and testing using a specialized language like SAS or R;
 2. Deployment of (some of the) prototypes in production using Java, C#, or C++.
- *There's a new kid on the block!* Python is a suitable language not only for research and prototyping but also building production systems.
 - No need to maintain two development environments when one will suffice.

Why *not* Python?

Performance

- Python (the language) has many run-time implementations.
 - [CPython](#) (most popular): C implementation;
 - [Jython](#): Java implementation;
 - [IronPython](#): C# implementation;
 - ...

Note that Python (the language) is not slow; some Python run-times (e.g., CPython) can be!

CPython

- Python source code is compiled to an intermediate language and then interpreted (i.e., executed) into a virtual machine.
- This is slower than native (assembly) code compiled directly from C/C++.
- It might also be slower than Java, which encapsulates some code optimization strategies (e.g., the Java JIT compiler).

Jython

- Python source code is compiled to Java intermediate language (*bytecode*) and then interpreted (i.e., executed) into the JVM.
- Performance is similar of that of a Java program.

IronPython

- It relies on the same .NET libraries and Common Language Infrastructure (CLI) as C#.
- Performance is similar of that of a C# program.

Other implementations

- Python can be directly translated to native code via PyREX, PyToC, and others.
- In this case, it will generally perform as well as C++.

Why *not* Python?

Concurrency

- Challenging language for building highly concurrent, multithreaded applications, particularly applications with many CPU-bound threads.
- This is due to the **global interpreter lock** (GIL), a mechanism which prevents the (CPython) interpreter from executing more than one Python instruction at a time.
- However, Python C extensions that use native multithreading (in C or C++) can run code in parallel without being impacted by the GIL, so long as they do not need to interact with Python objects.

Let's start getting our hands dirty!

[**Note:** *At this stage we assume our Python environment is correctly set up. We will dedicate a specific lecture to this later on.*]

The Python Interpreter

- The standard Python interpreter (CPython) runs a program by executing one statement at a time.
- It can be invoked on the command line with the `python` command:

```
> python
```

```
Python 3.6.2 |Anaconda custom (x86_64)| (default, Jul 20 2017, 13:14:59)  
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>>
```

The Python Interpreter

- The `>>>` is the *prompt* where you will type expressions.
- To exit the Python interpreter and return to the shell command prompt, you can either type `exit()` or press `Ctrl + D`.

Our first Python program: Hello World!

```
Python 3.6.2 |Anaconda custom (x86_64)| (default, Jul 20 2017, 13:14:59)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> s = "Hello World!"
>>> print(s)
Hello World!
>>> █
```

Our first Python program: Hello World!

- Instead of running it in *interactive* mode with the interpreter, a Python program can be executed directly from the command line. To do so:
 - Open a file with your favorite text editor (e.g., *vi*, *emacs*) and save it with `.py` extension (e.g., `hello_world.py`)
 - Put in it all the instructions you want to execute:

```
s = "Hello World!"  
print(s)
```

- Run the following command: `> python hello_world.py`

Installation and Setup

- There is no single solution for setting up Python and required add-on packages.
- I will try to give you detailed instructions on how to set those up on each major operating system.
- I recommend using the free [Anaconda](#) distribution provided by Continuum Analytics.
- Anaconda is offered in both Python 2.7 and 3.7 "flavours".
- This class assumes/encourage you to use Python 3.6 or 3.7 (more on Python 2.x vs. Python 3.x later).

Note: *Your system may come shipped with a default Python interpreter. Even so, we strongly encourage to follow the steps below in order to:*

- i) replicate the exact environment used in this class and*
- ii) practice with possible installation/configuration issues.*

Apple (OS X, macOS)

- Download the OS X Anaconda installer called something like `Anaconda3-5.3.0-MacOSX-x86_64.pkg`.
- Double-click the `.pkg` file to run the installer.
- When the installer runs, it automatically appends the Anaconda executable path (e.g., `/usr/local/anaconda3/bin/anaconda`) to your `/Users/$USER/.bash_profile` file.
- To verify everything is working, try launching `python` from the Terminal application.

Additional note for OS X and macOS

- Anaconda provides an easy way to install/setup Python as it comes with hundreds of useful packages bundled.
- In alternative (not recommended!), you may want to install Python interpreter alone (e.g., via `Homebrew`) and then install each package you need via `pip`, which is the default Python package manager.
- For those who are curious, please refer to this tutorial:
<http://docs.python-guide.org/en/latest/starting/install3/osx/>

GNU/Linux (1 of 3)

- Linux details will vary a bit depending on your distributions, e.g., Debian, Ubuntu, CentOS, and Fedora.
- Setup is similar to OS X with the exception of how Anaconda is installed.
- The installer is a shell script that must be executed in the terminal.

GNU/Linux (2 of 3)

- Depending on whether you have a 32-bit or 64-bit system, you will either need to install the x86 (32-bit) or x86_64 (64-bit) installer.
- You will then have a file named something similar to `Anaconda3-5.3.0-Linux-x86_64.sh`.
- To install it, execute this script with bash:

```
bash Anaconda3-5.3.0-Linux-x86_64.sh
```

GNU Linux (3 of 3)

- After accepting the license, you will be presented with a choice of where to put the Anaconda files.
- I recommend installing the files in the default location in your home directory, for example `/home/$USER/anaconda`.
- The Anaconda installer may ask if you wish to prepend its `bin` directory to your `$PATH` variable. You can always do this yourself by modifying your `~/ .bashrc` file as follows:

```
export PATH=/home/$USER/anaconda/bin:$PATH; source ~/ .bashrc
```

Windows (1 of 2)

- Download the Anaconda installer for Windows from <http://continuum.io/downloads>, an executable named like `Anaconda3-5.3.0-Windows-x86_64.exe`.
- Run the installer and accept the default installation location, which will either be `C:\Python37` (if run as an administrator) or `C:\Users\%USERNAME%\Anaconda3` if run as a normal user.
- If you had previously installed Python in this location, you may want to remove it first.

Windows (2 of 2)

- The installer will also ask you if you wish to make the Anaconda distribution the default Python on your system and whether to add it to your PATH environment variable (*recommended*).
- To verify that things are configured correctly:
 - Open a command prompt (Start Menu --> Command Prompt application);
 - Launch the Python interpreter by typing `python` and you should see a message that matches your version of Anaconda.

Installing Previous Version

- You may be interested in installing a specific version of Anaconda Python distribution (e.g., 3.6 instead of the latest 3.7)
- You can do that by clicking on the following [link](#)
- In this class we will be using Python 3.6, although the latest Python 3.7 (shipped with the latest Anaconda distribution) should work seamlessly.

Installing/Updating Packages

- To install additional Python packages which are not included in the Anaconda distribution you can either use `conda` (recommended) or `pip` as follows:
 - `conda install package_name`
 - `pip install package_name`
- A complete reference of available `conda` commands and comparison with `pip` can be found [here](#).

Python 2 vs. Python 3

- The first version of the Python 3.x line of interpreters was released at the end of 2008.
- It included a number of changes that made some previously-written Python 2.x code **incompatible**.
- It was the major "breaking" update since the very first release of Python in 1991.

Python 2 vs. Python 3

- In 2012, much of the scientific and data analysis community was still using Python 2.x because many packages had not been made fully Python 3 compatible.
- However, Python 2.x will reach its development end of life in 2020, and so it is no longer a good idea to start new projects in Python 2.x.
- A good reference to understand the main differences between the two versions can be found [here](#).

Text Editors and Integrated Development Environments (IDEs)

- Most text editors like [Atom](#) and [Sublime Text 3](#) have excellent Python support.
- However, when it comes to building/package more complex software, you may want to use a more richly-featured IDE like:
 - [PyDev](#) (free), a plugin built for the Eclipse platform;
 - [PyCharm](#) from JetBrains (paid for commercial users, free for open source developers);
 - [Python Tools for Visual Studio](#) (for Windows users);
 - [Spyder](#) (free), an IDE currently shipped with Anaconda.

Our Development Environment: Jupyter Notebook

- In this class, we will not use text editors nor fancier Python IDEs like [PyDev](#) (Python plugin for [Eclipse](#)).
- Instead, we will use [Jupyter Notebook](#) which is a web browser-based tool for fast developing, running and visualizing Python code.
- Let's see how to properly set up Jupyter Notebook's environment.

What's Jupyter Notebook?

Overview

- In a nutshell: A web application combined with a browser-based GUI which allows to create and share documents that contain live code, equations, visualizations as well as (formatted) text, exactly like **this one**.
- You can get (another) taste of Jupyter Notebook without having to install it on your local machine! Just open your web browser and go to <https://jupyter.org/try>.
- Main documentation of Jupyter can be found [here](#) (also available in [pdf](#)) and [here](#).

A Brief History of Jupyter and IPython

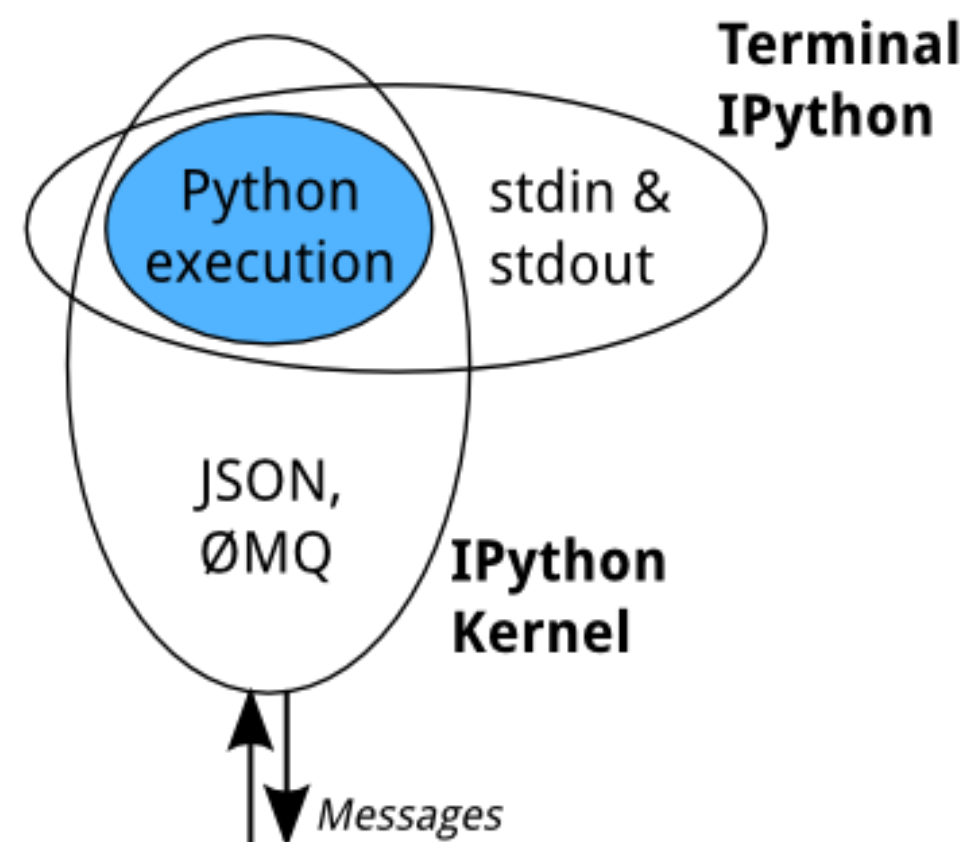
- The project [Jupyter](#) (as a whole) started as a spin-off project from [IPython](#),
- IPython was the first attempt towards developing an advanced and flexible framework for interactive, multi-language computing environment.
- Up to IPython 3.x, this was a monolithic project.
- Starting from IPython 4.0, the language-agnostic parts of the project have been moved out to an independent project codenamed, indeed, Jupyter.

IPython Today: Terminal and Kernel

- IPython itself now only refers to:
 - **Terminal IPython** as an *advanced* Read-Eval-Print-Loop (REPL) Python console (i.e., similar to the classic Python interpreter but with some additional features, such as tab autocompletion, color highlighting, etc.);
 - **IPython Kernel** that provides computation and communication (i.e., backend) capabilities to the frontend interfaces, such as (but not limited to) the (web) Notebook, the Qt console, the IPython console, and any other third-party interface.

IPython Today: Terminal and Kernel

- Frontends, like the Notebook or the Qt console, communicate with the IPython kernel using JSON messages sent over ZeroMQ sockets.
- The architecture of IPython (kernel and terminal) is depicted below:



IPython: Flexible Architecture

- IPython's architecture is designed to:
 1. Allow easy development of different frontends based on the same kernel.
 2. Support new languages in the same frontends, by developing kernels for those languages (e.g., **R**, **Julia**, **Haskell**, **C++**, etc.).

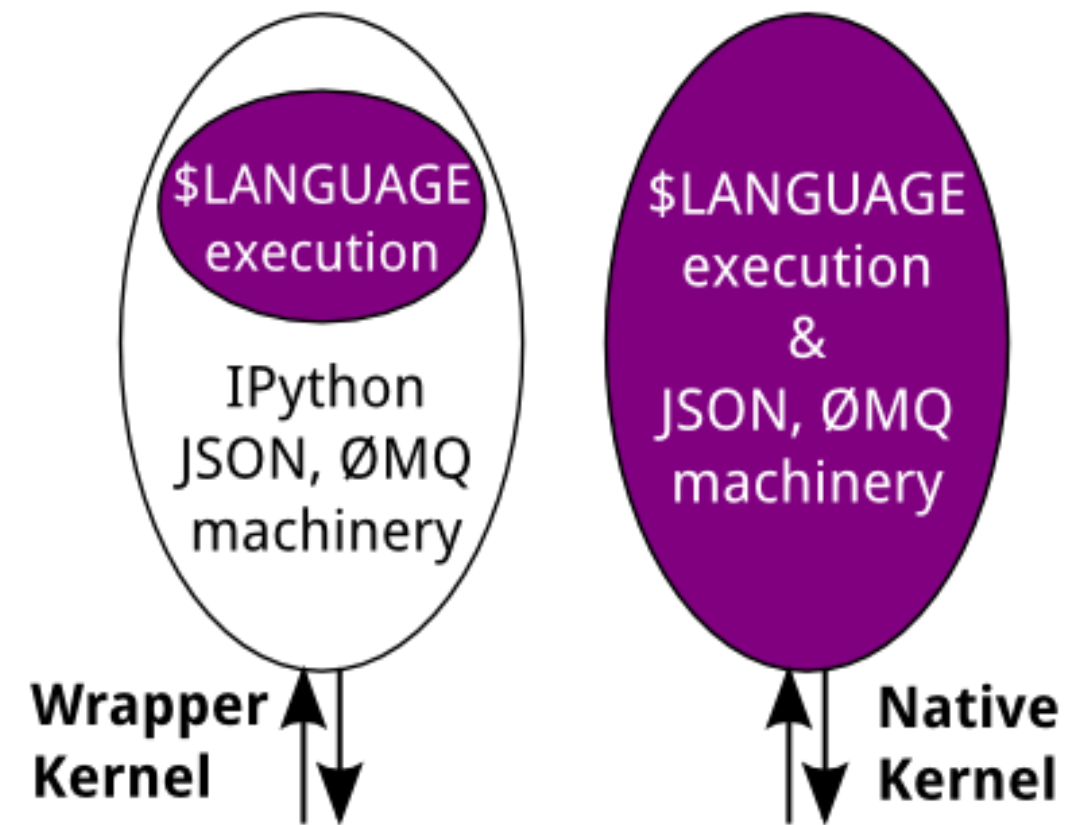
IPython: Adding Support for New Kernels

- There are **two** ways to develop a kernel for another language:
 1. **Wrapper kernels** reuse the communications machinery from IPython kernel, and implement only the core execution part.
 2. **Native kernels** implement execution and communications in the target language.

IPython: Wrapper vs. Native Kernels

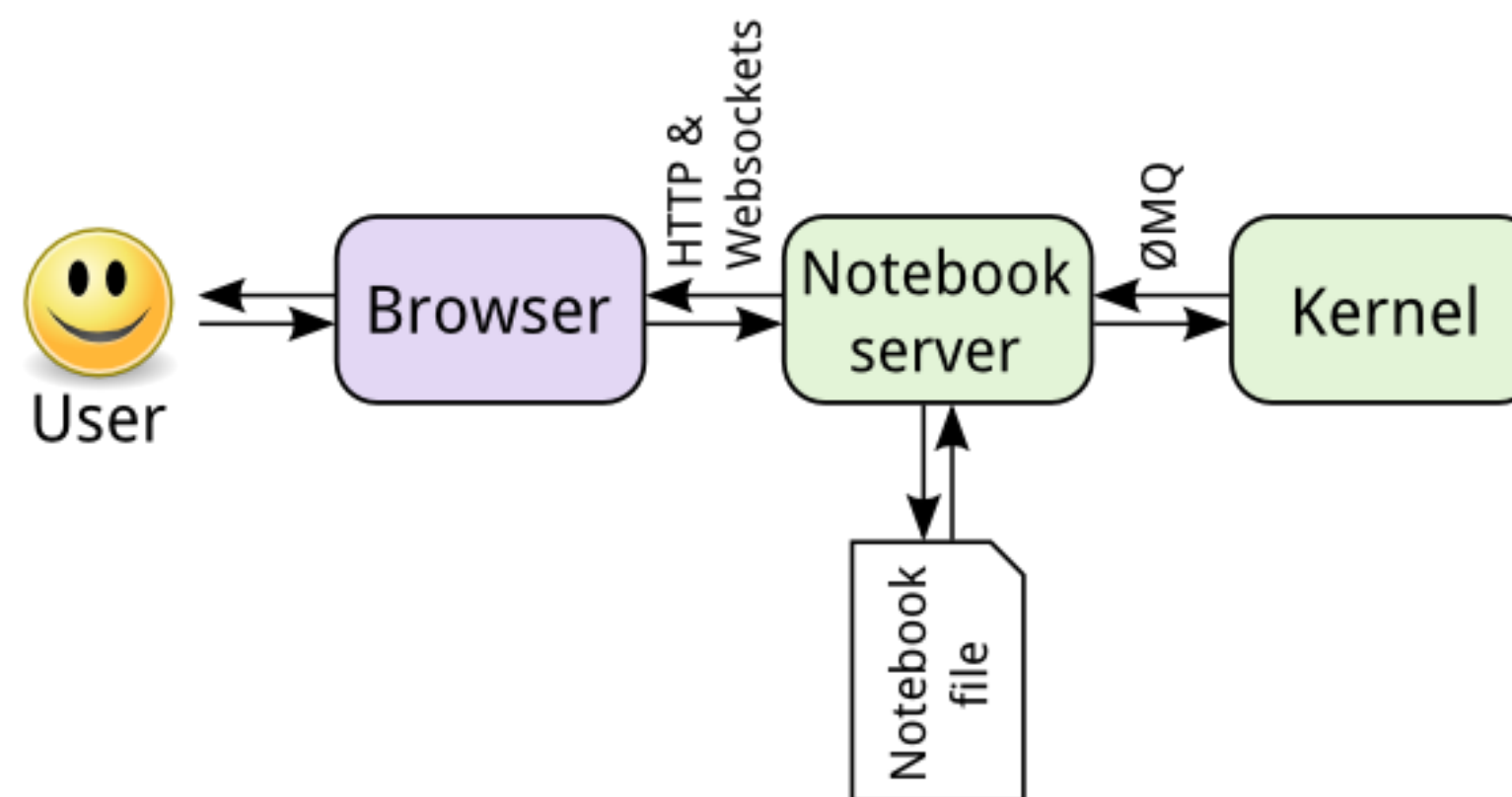
- **Wrapper kernels** are easier to write quickly for languages that have good Python wrappers, like [octave kernel](#), or languages where it's impractical to implement the communications machinery, like [bash kernel](#).
- **Native kernels** are likely to be better maintained by the community using them, like [Julia](#) or [Haskell](#).

IPython: Wrapper vs. Native Kernels



Jupyter Notebook's Overall Architecture

- Jupyter Notebook therefore refers to the combination of the IPython kernel (backend) with the web-based interface provided with Notebook (frontend).
- The following image describes the high-level architecture of Jupyter Notebook:



Jupyter Notebook's Workflow

- In addition to execute code, Jupyter Notebook stores code and output in an editable document called a *notebook*.
- Once saved, this is sent from your browser to the Notebook server, which saves it on disk as a JSON file with a `.ipynb` extension.
- Jupyter Notebook can be executed on a local machine without Internet access (i.e., via a web browser running on the same machine where the Notebook server is running), or on a remote server and accessed through the Internet.

[**Note:** We refer to "**Notebook**" as the Jupyter web application, whereas "**notebook**" stands for the JSON file representing any session stored on the Notebook server.]

Why Jupyter Notebook?

Overview

- Jupyter Notebook has gained momentum because represents the ideal tool for supporting Data/Machine Learning Scientists in their day-to-day work.
- There are now Jupyter Notebooks on numerous topics in many scientific disciplines. Here are a few examples of those:
 - [LIGO Gravitational Wave Data](#)
 - [Satellite Imagery Analysis](#)
 - [12 Steps to Navier-Stokes](#)
 - [Computer Vision](#)
 - [Machine Learning](#)

Literate Programming

- The reason for Jupyter's immense success is it excels in a form of programming called "*literate programming*".
- A style of software development pioneered by [Donald Knuth](#) which emphasizes a prose-first approach where exposition with human-friendly text is punctuated with code blocks.
- Cool for **research** and **teaching** (hope so!), especially for science.
- Jupyter Notebook allows and fosters the adoption of literate programming by embracing standard web technologies (i.e., mostly HTML5, JavaScript, and CSS).

How to Install Jupyter Notebook

Prerequisite: Python Interpreter

- While Jupyter runs code in many programming languages, the Python interpreter is a **requirement** (Python 3.3 or greater, or Python 2.7) for installing the Jupyter Notebook.
- The Python interpreter and Jupyter Notebook can be installed separately (in fact, this could be an option if you have already installed Python on your system).
- However, for a fresh install we **strongly recommend** using the [Anaconda](#) distribution.

Anaconda Distribution: Standard

- Anaconda is available for Windows, macOS, and Unix/Linux, and it comes in two open-source flavours: standard and [Miniconda](#)
- The standard version installs the Python interpreter (either one among the following versions: 2.7.13, 3.5.3, 3.6.1, or 3.7), `conda` (i.e., the package, dependency and environment manager)
- Plus, it installs **over 100** other pre-built and tested scientific and analytic Python packages.
- Among these packages are: [NumPy](#), [pandas](#), [SciPy](#), [matplotlib](#), [scikit-learn](#), and - indeed - Jupyter and IPython kernel.

Anaconda Distribution: Miniconda

- Instead, Miniconda is a very basic distribution which only contains the Python interpreter and `conda`.
- In any case, over 720 additional packages (as well as the support for other languages, like R or Scala) can easily be installed with `conda`.

Possible Alternatives

- Users who want to have more control over their systems may want to install the Python interpreter on their own (provided that either they do not have one installed on their machine or they want to install a different version of the Python interpreter already installed).
- macOS users should refer to the following [tutorial](#)
- Linux users should look into [this](#)
- Windows users, please follow this [tutorial](#)

Installing Jupyter Notebook

Following from the discussion above, there are essentially two ways of installing Jupyter Notebook:

- Using the Anaconda Python Distribution (**highly recommended**);
- Using [pip](#) (only for skilled users).

Installing Jupyter Notebook using Anaconda

- Once Anaconda is successfully installed on your machine, Jupyter Notebook (as well as any other required dependency, including the Python interpreter) is also installed, and ready to be executed.
- To actually run the Jupyter Notebook application, just open a shell and type the following command:

```
> jupyter notebook
```

Installing Jupyter Notebook using `pip`

- This is the most "*pythonic*" way of installing Jupyter Notebook and, due to that, it is better for experienced and skilled Python users.
- **IMPORTANT:** Jupyter installation requires Python 3.3 or greater, or Python 2.7. To provide support for Python 3.2 and 2.6, IPython 1.x - which included the parts that later became Jupyter - needs rather to be installed.

Installing Jupyter Notebook using `pip`

- Make sure you have the latest `pip` as follows:

```
# Python 3.3 or above
```

```
> pip3 install --upgrade pip
```

```
# Python 2.7.x
```

```
> pip install --upgrade pip
```

Installing Jupyter Notebook using `pip`

- Then, you can install Jupyter Notebook as follows:

```
# Python 3.3 or above  
> pip3 install jupyter
```

```
# Python 2.7.x  
> pip install jupyter
```

Optional: How to Install Additional IPython Kernels

- Installing the Jupyter Notebook will also install the corresponding IPython kernel. This allows working on notebooks using the Python programming language.
- More specifically, if Jupyter Notebook is running on Python 3 the corresponding IPython kernel (for executing Python 3 compliant code) will be installed.
- Vice versa, if Jupyter Notebook is running on Python 2 the IPython kernel installed will allow the execution of Python 2 code.

Optional How to Install Additional IPython Kernels

- Still, if you are running Jupyter Notebook on Python 3, you may want to set up an IPython 2 kernel like this:

```
> python2 -m pip install ipykernel
# 'Python 2' is the name shown in Notebook menus
> python2 -m ipykernel install \
--user --display-name "Python 2"
```

Optional How to Install Additional IPython Kernels

- Or using `conda`, create a Python 2 environment:

```
> conda create -n ipykernel_py2 python=2 ipykernel
# On Windows, remove the word 'source'
> source activate ipykernel_py2
# 'Python 2' is the name shown in Notebook menus
> python -m ipykernel install --user \
--display-name "Python 2"
```

- If you are running Jupyter on Python 2 and want to set up an IPython 3 kernel, follow the same steps as above, replacing `2` with `3`.
- Additional information on how to install/personalize kernels can be found [here](#).

Optional: How to Install Additional Languages (R, Julia, etc.)

- To run notebooks in languages other than Python, such as R or Julia, you will need to install additional kernels.
- The full list of available kernels can be found [here](#).

[**Note:** *We investigate how to integrate Jupyter Notebook with R later on this notebook.*]

How to Use Jupyter Notebook

Starting Up the Notebook Server

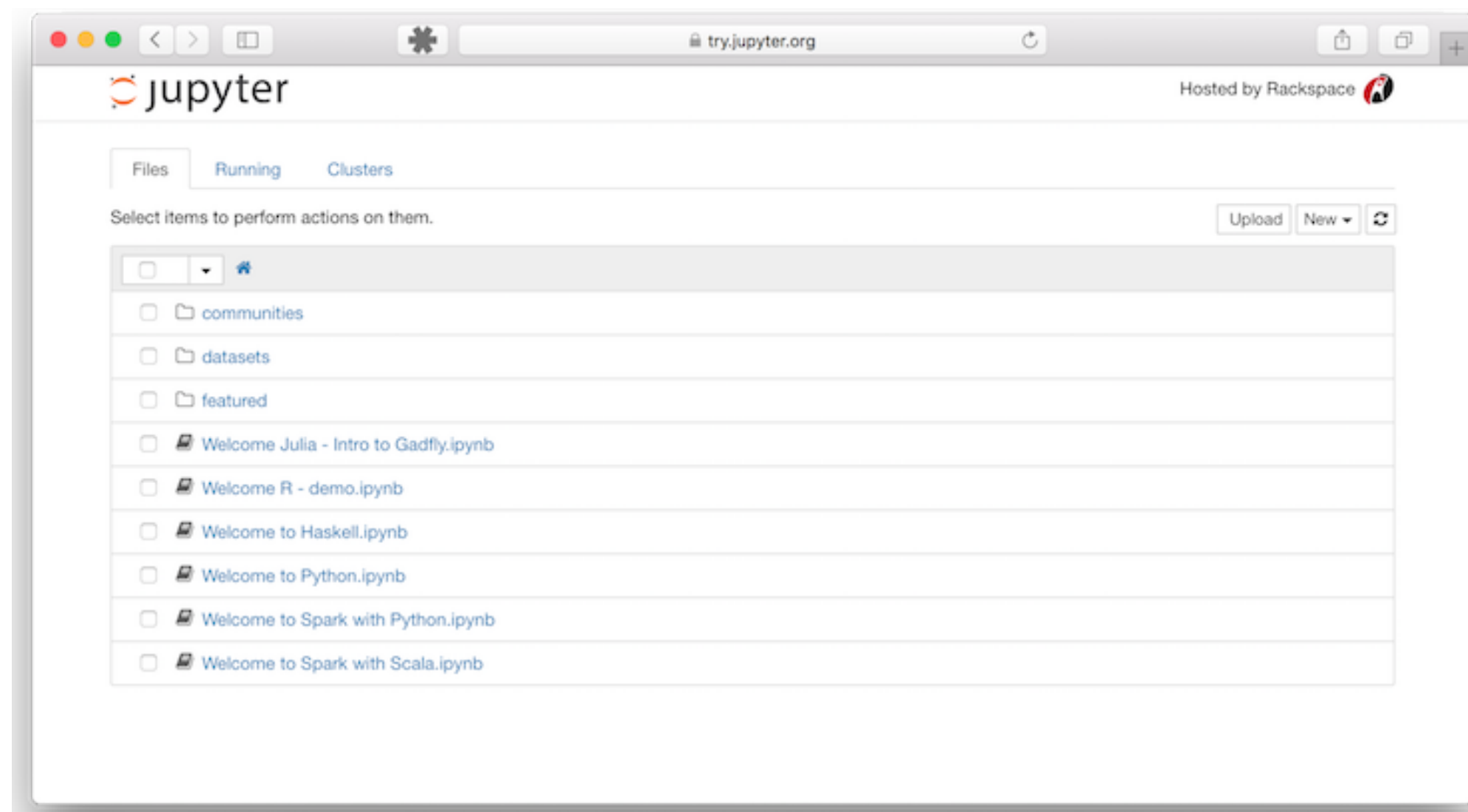
- To start the Jupyter Notebook Server, open a terminal window and type the following command:

```
> jupyter notebook
```

- This will print some information about the Notebook server to the stdout of your terminal, including the URL of the web application (by default, <http://localhost:8888>).

Starting Up the Notebook Server

- Once the notebook server is up and running, it will then open your default web browser to this URL showing the **Notebook Dashboard**, as follows:



Startup Options

- Any further Notebook Server's command line option (e.g., listening port) can be found [here](#).

Shutting Down the Notebook Server

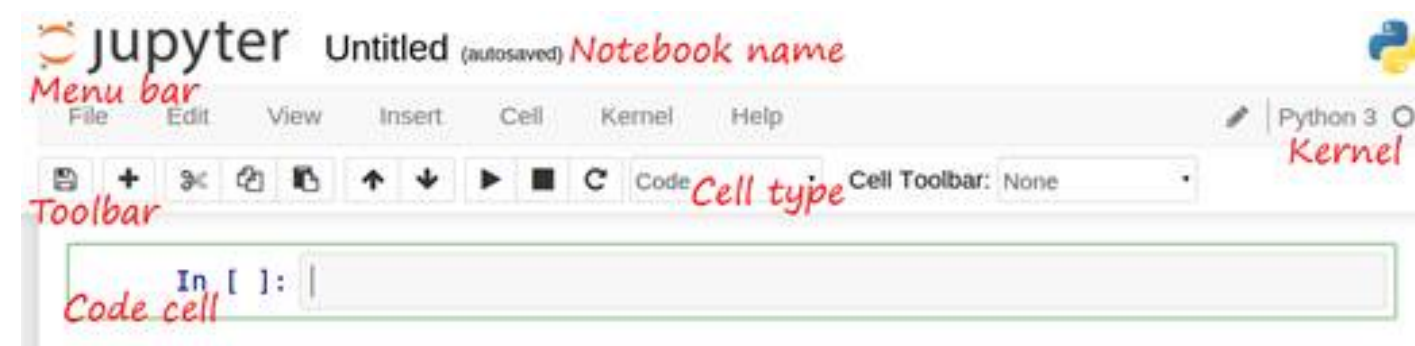
- Finally, to shutdown the Notebook Server just go back to the terminal window where you originally started it up and press **Ctrl + C** and confirm your choice typing **Y**.

The Notebook Dashboard

- The dashboard contains three tabs which are as follows:
 - **Files:** shows all files and notebooks in the current directory (i.e., the directory which you execute the `jupyter notebook` command from);
 - **Running:** keeps track of all kernels (processes) currently running on your computer;
 - **Clusters:** allows you launch kernels for parallel computing using the IPython's parallel computing framework.

The Notebook User Interface

- To create a new notebook, click on the **New** button (top-right corner of the dashboard) and select **Notebook (Python 3)**. A new browser tab opens up and shows the Notebook interface below:



The Notebook User Interface

- Here are the main components of the interface, from top to bottom:
 - The *Notebook name*, which you can change by clicking on it. This is also the name of the `.ipynb` file.
 - The *Menu bar* gives you access to several actions pertaining to either the notebook or the kernel.
 - To the right of the Menu bar is the *Kernel name*. You can change the kernel language of your notebook from the Kernel menu.
 - The *Toolbar* contains icons for common actions.

The Structure of a Notebook Cell

- There are two main types of cells: **Markdown** cells and **Code** cells.
 - A [Markdown](#) cell contains rich text. In addition to classic formatting options like **bold** or *italics*, we can add links, images, HTML elements, LaTeX mathematical equations, and much more (full specifications available [here](#)).
 - A Code cell contains code to be executed by the kernel. The programming language corresponds to the kernel's language.
- The type of a cell can be changed by first clicking on a cell to select it, and then choosing the cell's type in the toolbar's dropdown menu showing **Markdown** or **Code**.

Markdown Cell

```
### New paragraph

This is "rich" **text** with \[links\](http://ipython.org), equations:


$$\hat{f}(\xi) = \int_{-\infty}^{+\infty} f(x) e^{-i\xi x} dx$$


code with syntax highlighting:



```
python
print("Hello world!")
```



and images:


```

New paragraph

This is rich text with [links](#), equations:

$$\hat{f}(\xi) = \int_{-\infty}^{+\infty} f(x) e^{-i\xi x} dx$$

code with syntax highlighting:

```
print("Hello world!")
```

and images:

IP[y]: IPython
Interactive Computing

- The top panel shows the cell in **edit mode**, while the bottom one shows it in **render mode**.
- The edit mode lets you edit the text, while the render mode lets you display the rendered cell.

Example

This is a Markdown cell where we can write **bold**, *italic*, or any other HTML-formatted text.

We can add hyperlinks either using Markdown-specific syntax:

[\[link using Markdown syntax\]\(https://jupyter.readthedocs.io/en/latest/\)](https://jupyter.readthedocs.io/en/latest/)

or traditional HTML syntax:

[link using HTML syntax](https://jupyter.readthedocs.io/en/latest/)

Example

This is a Markdown cell where we can write **bold**, *italic*, or any other HTML-formatted text.

We can add hyperlinks either using Markdown-specific syntax:

[link using Markdown syntax](#)

or traditional HTML syntax:

[link using HTML syntax](#)

Example

Plus, we can also include mathematics in a straightforward way, using standard LaTeX notation: `\$...\$` for inline mathematics `\bar{X}=\frac{1}{N}\sum_{i=1}^N X_i`, and `\$\$...\$\$` for displayed mathematics:

`$$\bar{X}=\frac{1}{N}\sum_{i=1}^N X_i$$`

Not to mention, we can write down code with highlight syntax:

```
```python
print("Hello world!")
```
```

Example

Plus, we can also include mathematics in a straightforward way, using standard LaTeX notation: $\$...\$$ for inline mathematics $\bar{X} = \frac{1}{N} \sum_{i=1}^N X_i$, and $...\$$ for displayed mathematics:

$$\bar{X} = \frac{1}{N} \sum_{i=1}^N X_i$$

Not to mention, we can write down code with highlight syntax:

```
print("Hello world!")
```

Example

Finally, we can attach both local and remote images, again, using either Markdown-specific or HTML syntax.

local/Markdown-specific:

```
<center>![This is a local image using Markdown syntax](./img/jupyter_logo.svg)</center>
```

local/HTML:

```
<div>
```

```
<center></center>
```

```
</div>
```

Example

Finally, we can attach both local and remote images, again, using either Markdown-specific or HTML syntax.

local/Markdown-specific:



local/HTML:



Example

remote/Markdown-specific:

```
<center>![This is a remote image using Markdown syntax]
(https://jupyter.readthedocs.io/en/latest/_static/_images/jupyter.svg)</center>
```

remote/HTML:

```
<div>
<center></center>
</div>
```

Example

remote/Markdown-specific:



remote/HTML:



Code Cell

In [1]: `import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
print("Hello world!")
plt.imshow(np.random.rand(20, 20), interpolation='none');
from IPython.display import display_html
from IPython.html.widgets import FloatSlider
display_html('<table><tr><td>some</td><td>table</td></tr></table>', raw=True)
FloatSlider(value=70)`

Widget area

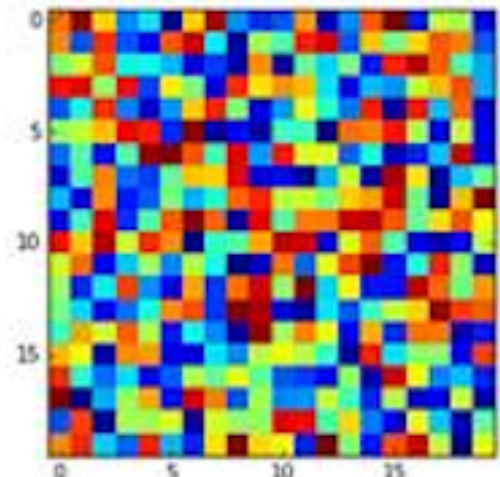
Output area

Standard output

Error output

Rich output

some table



Code Cell

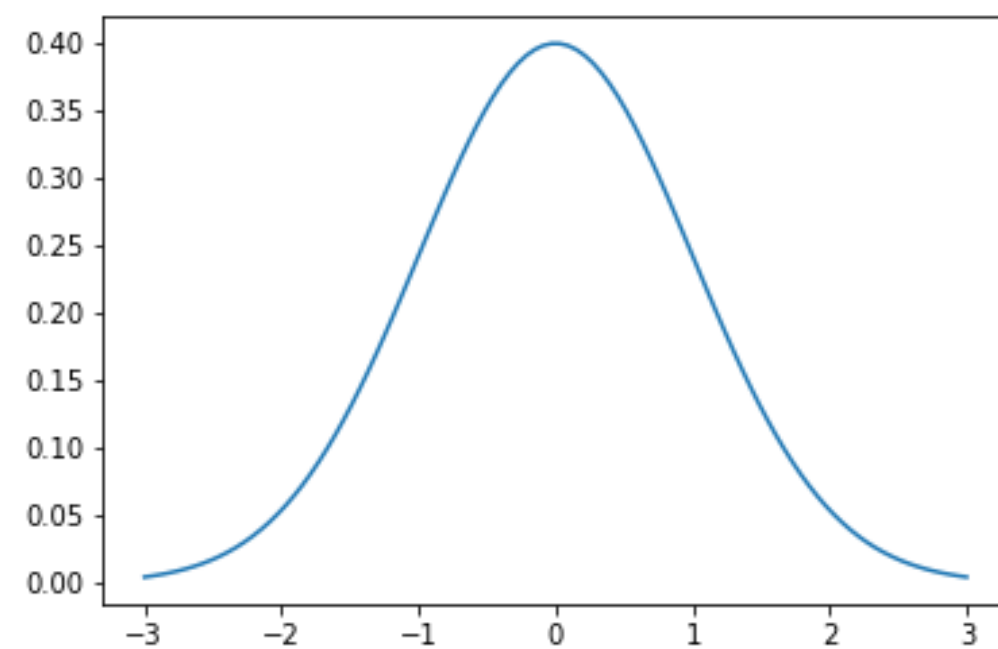
- The *Prompt number* shows the cell's number. This increases every time you run the cell (possibly unordered as you can run cells arbitrarily).
- The *Input area* contains a multiline text editor that lets you write code with syntax highlighting.
- The *Widget area* may contain graphical controls.
- The *Output area* can contain multiple outputs, specifically:
 - Standard output (text in black)
 - Error output (text with a red background)
 - Rich output (in the screenshot above, an HTML table and an image)

In [1]:

```
# This is a code cell  
def add(a, b):  
    return a + b  
add(5, 4)
```

Out[1]: 9

```
In [1]: 1 matplotlib inline
2
3 import matplotlib
4 import math
5 import matplotlib.pyplot as plt
6 import matplotlib.mlab as mlab
7 import numpy as np
8
9 mu = 0
10 variance = 1
11 sigma = math.sqrt(variance)
12 x = np.linspace(mu-3*variance, mu+3*variance, 100)
13 _ = plt.plot(x, mlab.normpdf(x, mu, sigma))
```



Modal Interface: Edit vs. Command Mode

- The Notebook implements a modal interface similar to some text editors such as `vim`.

Edit Mode

- Use the **edit mode** to write code (the selected cell has a green border, and a pen icon appears at the top right of the interface).
- Click inside a cell or press `Enter` to enable the edit mode for this cell (you need to double-click with Markdown cells).

Command Mode

- Use the **command mode** to operate on cells (the selected cell has a light-blue border, and there is no pen icon).
- Click outside the text area of a cell to enable the command mode (or press the **Esc** key).
- Type **H** (when in command mode) to show the list of available keyboard shortcuts.

Keyboard Shortcuts (available in *both* modes)

- Here are a few keyboard shortcuts that are always available when a cell is selected:
 - **Ctrl + Enter**: run the cell
 - **Shift + Enter**: run the cell and select the cell below
 - **Alt + Enter**: run the cell and insert a new cell below
 - **Ctrl + S** (Windows/Linux) or **Cmd + S** (Mac): save the notebook

Keyboard Shortcuts (available in the *edit* mode)

- In the edit mode, you can type code as usual, and you have access to the following keyboard shortcuts:
 - **Esc**: switch to command mode
 - **Ctrl + Shift + -**: split the cell

Keyboard Shortcuts (available in the *command* mode)

- In the command mode, keystrokes are bound to cell operations. Don't write code in command mode or unexpected things will happen! For example, typing **DD** in command mode will delete the selected cell! Here are some keyboard shortcuts available in command mode:
 - **Enter**: switch to edit mode
 - **Up** or **K**: select the previous cell
 - **Down** or **J**: select the next cell
 - **Y/M**: change the cell type to Code cell/Markdown cell
 - **A/B**: insert a new cell above/below the current cell
 - **X/C/V**: cut/copy/paste the current cell

Keyboard Shortcuts (available in the *command* mode)

- (Continue):
 - **DD**: delete the current cell
 - **Z**: undo the last delete operation
 - **Shift + M**: merge multiple cells into a single one
 - **H**: display the help menu with the list of keyboard shortcuts
- Remember that some keyboard shortcuts may be different on different platforms (Windows/Linux and Mac).
- Please, have a look at [28 Jupyter Notebook tips, tricks and shortcuts](#) and [Jupyter \(IPython\) notebooks features](#) for any further details.

Jupyter Notebook (Advanced)

Jupyter and `conda` for R

- We already said that Jupyter began with Python but now has kernels for 50 different languages, including R, which is widely used by statisticians.
- [IRKernel](#) is the native R kernel for Jupyter.
- Therefore, those who are more familiar with R than Python still can benefit from Jupyter.

Setup Jupyter for R: Get IRKernel

- The first step to do is to use `conda` package manager to download "[R Essentials](#)" bundle.
- This bundle contains IRKernel and over 80 of the most widely-used R packages for data science, including `dplyr`, `shiny`, `ggplot2`, `tidyr`, `caret`, and `nnet`.

Setup Jupyter for R: Install `r-essentials`

- Provided that `conda` is already available on your system, "R Essentials" can be installed in the current environment by issuing the following shell command:

```
> conda install -c r r-essentials
```

Setup Jupyter for R: Install `r-essentials` on a dedicated environment

- Alternatively, if you want to create a dedicated environment just for R (e.g., called `R-env`), you can type as follows:

```
> conda create -n R-env -c r r-essentials
```

Setup Jupyter for R: Install `r-essentials` on a dedicated environment

- This will require you to **activate** that environment in order for Jupyter Notebook to use it with the following command:

```
> source activate R-env
```

- Then, you can start Jupyter Notebook as usual:

```
> jupyter notebook
```

Share Jupyter Notebooks

- Remember that notebooks are just JSON documents that contain text, source code, rich media output, and metadata. Each segment of the document is stored in a cell.
- Ideally, you don't want to go around and share your notebooks with others as JSON files.

Share Jupyter Notebooks

- Under the *File* menu, Jupyter gives you the option to download your notebook as a Notebook (`.ipynb`), Python script (`.py`), HTML (`.html`), Markdown (`.md`), reStructuredText (`.rst`), LaTeX (`.tex`) or PDF (`.pdf`) file.
- Alternatively, you can use the `nbconvert` command to convert your notebook document file to another static format.

Converting Jupyter Notebook to Slides

- Jupyter can convert a notebook into an online slide deck for talks and tutorials (like this one!).
- To convert a notebook into a [reveal.js](#) presentation, go to the *View* menu and set *Cell Toolbar* to *Slideshow*.
- Organize the cells of your notebook into *slides* and *subslides*.

Converting Jupyter Notebook to Slides

- Convert it to the slide format using `nbconvert` as follows:

```
> jupyter nbconvert {NOTEBOOK_NAME}.ipynb --to slides --post  
rve
```

- This will generate a `{NOTEBOOK_NAME}.slides.html` file in the same directory of your notebook file
- You can open it on a new tab of your system default browser at `http://localhost:8888/{NOTEBOOK_NAME}.slides.html#/`

Converting Jupyter Notebook to Slides (alternative way)

- Install `RISE` plugin as a Jupyter Notebook extension as follows:

```
> conda install -c damianavila82 rise
```

- Other possible ways of installation can be found [here](#).
- This deck has been generated with `RISE`.


```
In [3]: # The following piece of code will generate a JSON configuration (liverreveal.json) file  
# under your Jupyter configuration path (e.g., /usr/local/anaconda3/etc/jupyter/nbconfig)  
from traitlets.config.manager import BaseJSONConfigManager  
path = "path/to/your/jupyter/nbconfig" # e.g., "/usr/local/anaconda3/etc/jupyter/nbconfig"  
cm = BaseJSONConfigManager(config_dir=path)  
cm.update('liverreveal', {  
7     'theme': 'sky', # change the default slide's background theme  
8     'transition': 'zoom', # zooming when transitioning from one slide to the other  
9     'start_slideshow_at': 'selected', # start with the selected cell of the notebook  
10    'width': 1280, # slide's width  
11    'height': 768, # slide's height  
12    'scroll': True # vertical scroll (default is False)  
13})
```

```
Out[3]: {'height': 768,  
        'scroll': True,  
        'start_slideshow_at': 'selected',  
        'theme': 'sky',  
        'transition': 'zoom',  
        'width': 1280}
```

Printing Jupyter Notebook

At least, there are 3 ways to print out your notebook and each of them needs to transform the original notebook file (`{NOTEBOOK_NAME}.ipynb`) into a `{NOTEBOOK_NAME}.pdf` file.

Method 1: LaTeX

- Use the *File --> Download as --> PDF via LaTeX (.pdf)*
- This will (attempt to) produce a PDF file which mimics the original notebook layout by taking advantage of the `pdflatex` tool.
- You might get some errors/unexpected behavior due to LaTeX issues on any of the cells of your notebook.

Method 2: `nbconvert` to HTML

An alternative (and possibly better) solution could be the following:

- Transform your notebook into a "static" HTML file using `nbconvert`:

```
> jupyter nbconvert {NOTEBOOK_NAME}.ipynb --to html
```

- This will generate a `{NOTEBOOK_NAME}.html` file in the same directory of your notebook file.
- Finally, you can open this HTML file with your favourite web browser and use the "Print PDF" dialog to save it as a pdf file.

Method 3: `nbconvert` to slides

If you have already transformed your notebook to slides using `nbconvert` and you want to get a pdf-printable version of them here is one possible trick:

- Replace the original address

```
http://localhost:8888/{NOTEBOOK_NAME}.slides.html#/
```

with the following

```
http://localhost:8888/{NOTEBOOK_NAME}.slides.html?
```

```
print-pdf
```

Note however that due to some issues with `reveal.js` this might lead to sloppy results (e.g., text overlapping, wrong page breaks, etc.)

Method 4: DeckTape + RISE slides

- Instead, a much better result is given by creating slideshow with RISE and use [DeckTape](#) to transform slides into pdf.
- First of all, install RISE and DeckTape as follows:

```
> conda install -c damianavila82 rise # Install RISE
> npm install -g decktape # Install DeckTape
> npm update -g decktape # Update DeckTape
```

Method 4: DeckTape + RISE slides (conversion)

- Assuming you have launched your notebook at

`http://localhost:8888/{NOTEBOOK_NAME}.ipynb` and let

`{TOKEN}` be the token associated with your notebook then run the

following command:

```
> decktape rise http://localhost:8888/{NOTEBOOK_NAME}.ipynb?token={TOKEN} \
--size 1920x1080 path/to/outputfile.pdf
```

Jupyter Notebooks for Data Science Teams: Best Practices

- Jonathan Whitmore wrote in his [article](#) some practices for using notebooks for data science, such as the following:
 - Use two types of notebooks for a data science project, a **lab** notebook and a **deliverable** notebook; individuals control the lab notebook, whilst the whole team manage the deliverable.
 - Use some type of versioning control (e.g., `GitHub`). Don't forget to commit also the HTML file if your version control system lacks rendering capabilities.
 - Use explicit rules on the naming of your documents.

Learn From The Best Notebooks (not exhaustive list!)

- Notebooks are also used to complement books, such as the *Python Data Science Handbook*. You can find the notebooks [here](#).
- A report on a [Kaggle](#) competition is written down in [this blog](#), generated from a notebook.
- This [matplotlib tutorial](#) is a great example of how well a notebook can serve as a means of teaching other people topics such as scientific Python.
- Lastly, make sure to also check out [The Importance of Preprocessing in Data Science and the Machine Learning Pipeline](#) tutorial series that was generated from a notebook.

By the end of Week 1

- Install the full Anaconda distribution on your own system, using the steps described before.
- Familiarize with the Python shell interpreter (i.e., execute `python` from the command line).
- Familiarize with Jupyter Notebook (i.e., execute `jupyter notebook`, play with code/markdown cells, edit/command mode, etc.)