Setting Up

To run Python programs all we need is a Python interpreter and a text editor. In the next section you will learn how to install the reference Python interpreter. Then we will discuss what IDEs are available for Python.

Installing Python

The reference Python implementation is CPython. We will use the 3.6.x version. The Python 2 version is being discontinued and will soon not be supported anymore, so do not even bother with it.

To install, go to this download page for Python 3.6.8. On the bottom of the page you will find the links to the installer for different operating systems. If you are running Windows or MacOS, select the appropriate installer (64-bit) and download it. Before installing Python read the section bellow for your operating systems.

Windows

Run the installer, then:

- In the installation check the boxes: `Install launcher for all users` and `Add Python 3.6 to PATH`
- Then click on `Install Now`
- Click on `Disable path length limit`
- Finish installing and close the installer

Open the Windows `Power Shell`. Notice that the power shell is not the same program as the command prompt. The command prompt is usually a black screen, while the Power Shell will open up a blue screen.

On the `Power Shell` type:

```
# Show where Python was installed
Get-Command py
# Verify that the version of Python is in fact 3.6.x
py --version                    # Should print 3.6.x
```

The last command should print the correct version of Python that was just installed. If it does not, then you probably have already installed some other version of Python. If the version is a recent one, 3.6.x or 3.7.x or higher, then it should be fine. If it is not, then you will need to either uninstall the previous version and then reinstall Python 3.6.8, or use a virtual environment (discussed in a section below).

You can now launch the Python interpreter by running:

```
py
```

This will launch Python and you can now type Python commands to have them evaluated just in time.

Notice that Windows refers to Python as `py`, but all other operating systems use the name `python` or `python3`. For this reason, moving on, we will refer to Python commands that should be run on the Power Shell with `python` instead of `py`. But to make the command work in Windows you should write `py` instead of `python`.

MacOS

Run the installer with all defaults. After installing, open `Terminal` and verify the installation:

```
# Show where Python was installed
which python
# Verify that the version of Python is in fact 3.6.x
python --version
```

MacOS usually comes with a Python 2.x preinstalled, so the command above might return something like `Python 2.7.10`. However, you can still access the Python you just installed with the command `python3` or even `python3.6`. To verify this is the case, run the following on `Terminal`:

```
python3 --version
```

And you should get back `Python 3.6.8`.

You can now launch the Python interpreter by running:

```
python3
```

This will launch Python and you can now type Python commands to have them evaluated just in time.

Moving on, whenever you see a command for `Terminal` starting with `python`, please make the substitution to `python3`.

GNU/Linux

To install Python on GNU/Linux, open `Terminal` and run:

```
# Update source list
sudo apt-get update
# Install Python 3.6
sudo apt-get install python3.6
sudo apt-get install python3-pip
```

Now, check if we got the correct version installed by typing the following:

```
python --version
```

On GNU/Linus you may already have a Python 2 installed, so Python 3.6 would be available as `python3` or even `python3.6` instead. If the command above does not return `Python 3.6.x`, then try the following:

```
python3 --version
# or
python3.6 --version
```

And you should see `Python 3.6`.

You can now launch the Python interpreter by running:

```
python3
```

This will launch Python and you can now type Python commands to have them evaluated just in time.

Moving on, whenever you see a command for `Terminal` starting with `python`, please make the appropriate substitution to `python3` or `python3.6`.

Python Package Installer

Now, we need to install the Python Package Installer, also known as `pip`, which is responsible for downloading and installing additional packages that we might require. To do so, run the following in the `Terminal` (or `Power Shell`):

```
python -m pip install -U pip
```

Remember to make the appropriate substitution of `python` so that you are using Python 3 and not Python 2.

The command above will install and upgrade `pip` to the latest version. You can now use `pip` to install packages.

The packages installed using `pip` are fetched from a central repository called the Python Package Index (PyPI, `https://pypi.org`). These packages are installed in a folder named `site-packages`, and the packages are local to each version of Python (more about this on the section about virtual environments).

Packages can be installed with the command `pip install <package_name>`, and uninstalled with `pip uninstall <package_name>`. In some systems, it is not possible to call `pip` directly, so you should run instead:

```
python -m pip install <package_name>
```

The command `pip list` lists all installed packages and their version numbers. The command `pip list --outdated` shows installed packages that are outdated. Outdated packages can be updated with `pip install --upgrade <outdate_package_name>`. The command `pip show <package_name>` gives a detailed description of the installed package, including where it is installed, the version number and dependencies.

An important feature of `pip` is that it can install packages listed in a file automatically. Why is this useful? Let's say you are working on a research project, and end up installing 20 different packages. Then, you need to share your code with a co-author or you need to run your code on a faster computer. Then you need to find out what were all of the 20 packages you installed and what were their versions, since using very different versions of packages could break your code. Fortunately, `pip` makes this task easy:

```
pip freeze > requirements.txt
```

The command above will create a text file containing a list of all the packages you have installed and their versions. Each line will contain a single package and the version number. You can send this file to your co-author or new computer and run:

```
pip install -r requirements.txt
```

The command will install all of the packages in the same versions you were originally using.

Text Editors and Integrated Development Environments (IDEs)

To write Python programs we need to write commands in a text file ending in `.py`. Then we tell Python to execute the commands in that file. In order to help us write those files, it is useful to have a text editor that has syntax highlighting and auto completion of certain terms.

Below is a short list of possible text editors that you can use:

- Editors/IDEs that are easy to setup and use:
  - Any text editor, like TextEdit or Notepad. The caveat with these editors is that there is no syntax highlighting or other utilities.
  - IDLE: comes installed with Python. Provides a simple editor and Python shell.
  - Anaconda is a full IDE that comes with many packages preinstalled (it also comes with Jupyter, mentioned in the list below). It is open source and free to use. Most beginners to Python use this distribution. It provides an environment similar to Matlab or R, with a text editor and a REPL. Installing Anaconda will install Python in a virtual environment so that it is separate from the Python in your system. More about virtual environments in the sections below.
- Editors/IDEs that are medium difficulty to setup and use:
  - Jupyter Notebook: it is a web application for creating documents that mix text, images and code. It is open source and free to use. We will use this IDE to code during most of the lectures.
  - Visual Studio Code: an IDE by Microsoft. You can add support for Python within the program. It is free to use and can be customized.
  - Atom: an IDE created by people at Github. You can add support for Python installing packages such as this. It can be customized with JavaScript.
  - Sublime: paid text editor. Can be used for Python with plugin.
- Editors/IDEs that are hard to setup and use:
  - GNU Emacs: a customizable text editor. Can be made into an IDE for Python with packages such as Elpy.

  - Vim: a customizable text editor. Can be made into an IDE for Python by following this tutorial.

If you write a lot of Python code, you should think about learning how to use Emacs or Vim, since they are incredible text editors (but with a steep learning curve). Anaconda is a very good IDE for beginners and comes with most of the packages we will use preinstalled, including Jupyter. Jupyter Notebook is great for sharing code with others during presentations (or discussions with faculty). During the lectures we will use Jupyter Notebook as our Python IDE, but when you are coding by yourself you may want to use something else. Feel free to use whatever editor you are most comfortable with.

Virtual Environments

So far we have installed Python 3.6.8 in our computer and also the package Jupyter Notebook. You probably have run into the issue that another version of Python was already installed on your system, most likely Python 2. And that to access the correct version of Python we had to be careful and use `python3` or `python3.6`. To avoid having to deal with these issues we will create something called a virtual environment.

To understand what a virtual environment is, we need to understand how your computer finds a program when you type `python` in the `Terminal`. When we tell the terminal to run a program, it tries to find the program in a list of folders. This list of folders is stored in a variable named `PATH`. You can see this list by typing the following in the `Terminal`:

```
1 echo $PATH
```

In `Power Shell`:

```
1 $Env: Path
```

You will get a list like this:

```
1 /usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

The folders are separated by colons, and the search for the program follows the order of the folders in the list. In this case, if we type `python`, we will search for the program `python` in the folder `/usr/local/bin`, since my Python 3.6.x is installed in that folder it will be found and executed. This is why, in my computer when I type `which python`, I get back `Python 3.6`. Now, if I did not have Python 3.6 installed, then there would be no `python` in the folder `/usr/local/bin` and the search would fail. We would then search for `python` in `/usr/bin` where we would find it. In my computer, the `python` inside `/usr/bin` is the Python installed with the operating system (Python 2.7.10).

What would happen if we switched the order of folders in `PATH` to the following:

```
1 /usr/bin:/usr/local/bin:/bin:/usr/sbin:/sbin
```

And then typed:

```
1 which python                    # Get-Command py in Windows
```

What should be the result of the command above? It would print out `Python 2.7.10`, since it would search for `python` now starting in the `/usr/bin` folder instead of `/usr/local/bin`.

Therefore, changing the order of folders in `PATH` changes where the system will search for programs to run.

What a virtual environment does is temporarily modify the list of folders in `PATH`, so that if we type `python` we get the Python version we expect to get, no matter where it is.

Creating a Virtual Environment for Python

There are many ways to create a virtual environment, here we will use the package `virtualenv`. This package creates an isolated Python environment in a folder. We can activate this environment so that `python` points to the Python installed in that folder. This also means that packages installed with `pip` will be installed for the Python in that folder. This isolates Python in a way that is easy to replicate and control.

Install `virtualenv` with `pip`:

```
python3 -m pip install virtualenv
```

Next, we will call `virtualenv` with two arguments. The first argument is the path to the folder where we want to create the virtual environment. The second argument is the python version we want to install in that environment. Run the following:

```
virtualenv ~/Desktop/py36 --python=python3.6
```

If you are a Windows user and you get an error message about "Execution Policy", then you should read the blue `Note` box in this page.

At this point `virtualenv` has created a folder `py36`. Its contents are:

```
py36/
|-- bin/      # contains python and pip binaries, and
   activate script
|-- include/  # C code used by Python
|-- lib/      # python packages: standard library and
   downloads
    |-- python3.6/
        |-- site-packages/ # packages installed with pip go
           here
```

If you are a Windows user, then you should see similar folders and a folder named `Scripts`.

To actually make the virtual environment active, execute the following:

```
source ~/Desktop/py36/bin/activate
```

In `Power Shell`:

```
. ~\Desktop\py36\Scripts\activate.ps1
```

This script will activate the virtual environment, making the necessary changes to the `PATH` variable, so that typing `python` just works.

We can check that we are calling the right Python by running:

```
which python                    # returns ~/Desktop/py36/bin/
   python
```

```
python --version                    # returns Python 3.6.8
echo $PATH                          # notice what is the first
   folder in the list!
```

Now we can run our Python code, install packages and everything related to Python will be contained in the `py36` folder. We localized Python 3.6 and any packages we install to the `py36` folder. Because we are using the virtual environment, we do not need to remember to make the appropriate substitution of `python` for `python3` or `python3.6` anymore.

After you are done working with Python 3.6 you can simply run `deactivate` to exit the virtual environment (or just close the terminal).

Notice that your programs do not need to necessarily be on the `py36` folder. That folder simply contains Python and packages that we will use when the environment is active. The IDE Anaconda creates a virtual environment with Python and other packages under the hood, basically replicating the process we above.

Installing Jupyter Notebook

We will now install Jupyter Notebook, which is the web browser based IDE we are going to use during the lectures. To do so, first activate the virtual environment so that we install Jupyter for the right version of Python.

```
source ~/Desktop/py36/bin/activate
```

To install Jupyter Notebook we can use `pip`. Run the following in a `Terminal` (or `Power Shell` for Windows users):

```
pip install jupyter
```

Now you have installed Jupyter Notebook. To run the notebook, simply type:

```
jupyter notebook
```

This will start the jupyter notebook server on your terminal and then launch a browser with the Jupyter Notebook application. There you can create a new file and start coding. To shutdown the notebook just go back to the Terminal and hit Ctrl-C twice.

Preparing For the Next Lectures

At the beginning of each lecture, the first thing you should do is activate the virtual environment:

```
source ~/Desktop/py36/bin/activate
```

We will also create a folder to hold all of our lecture notes:

```
# Create a directory to hold lectures
mkdir ~/Desktop/python_course
cd  ~/Desktop/python_course
# in Bash this can be done in one line
mkdir ~/Desktop/python_course && cd "$_"
```

```
# $_ is a special symbol that captures the arguments of the last command
# we add quotes around it to deal with possible spaces in the name
```

Finally, you should launch Jupyter Notebook:

```
# Launch jupyter notebook
jupyter notebook
```

At each lecture we will code in Python using Jupyter notebooks, one for each topic. When you download Jupyter notebooks from the course website, you will get files ending in `.ipynb`. Put these files in the folder `python_course` you just created, and when you launch `jupyter notebook` from that folder, you should see those files in your browser. If you click on a file, a new tab will open and you can start coding.

Jupyter Notebook Basics

Jupyter notebook is a software (web application) that facilitates sharing interactive documents, referred to as jupyter notebooks. We will use these notebooks to learn Python. We can use jupyter notebooks to write code, execute it and interact with its outputs. Next, we discuss the basics of Jupyter Notebook.

Launch Jupyter Notebook in the `python_course` folder:

```
source ~/Desktop/py36/bin/activate
cd ~/Desktop/python_course
jupyter notebook
```

Your browser should open a new tab with Jupyter Notebook (see Figure 1).

Create a new notebook by clicking on the `New` button and then selecting `Python 3` (see Figure 3).

The notebook will open in a new tab (see Figure 3).

The box you see on the screen is called a `cell`. There are two types of cells: code cells and markdown cells.

On code cells you can type Python code. Go ahead and type `print("Hello World!")`. We can execute the code by going on the menu `Cell` and clicking on the button `Run Cells`. This will run the Python code you just typed, and the output of this code, if any, will be displayed right below the code (see Figure 4). Alternatively, you can run the code with the shortcut `Ctrl-Enter`. You can also run the code in the cell with the shortcut `Shift-Enter`. Pressing `Shift-Enter` will run the code, output its results, and create a new code cell below the one you just executed.

Notice that after hitting `Shift-Enter`, you get a new code cell, and its left side has a blue vertical bar (see Figure 5).

When the cell is in blue, we are in `Command Mode`. This means we can use some shortcuts to move between cells and other functionality. If you hit `Enter`, then you will go inside the cell and you are now in `Edit Mode`. Notice that the left side of the cell is now green. In this mode, you can type text inside the cell, and other shortcuts are available. You can exit `Edit Mode` to `Command Mode` by hitting `ESC`.

We can change a code cell into a markdown cell by hitting `m` (for markdown) while on `Command Mode`. Notice that now the word `In [ ]:` disappears from the cell. The

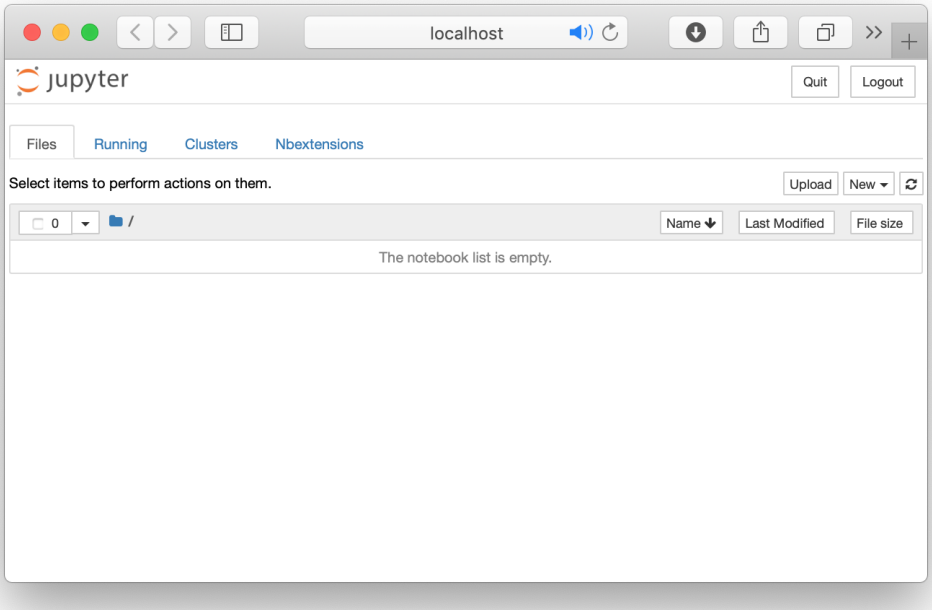

Figure 1: Launching Jupyter Notebook for the First Time on an Empty Folder.

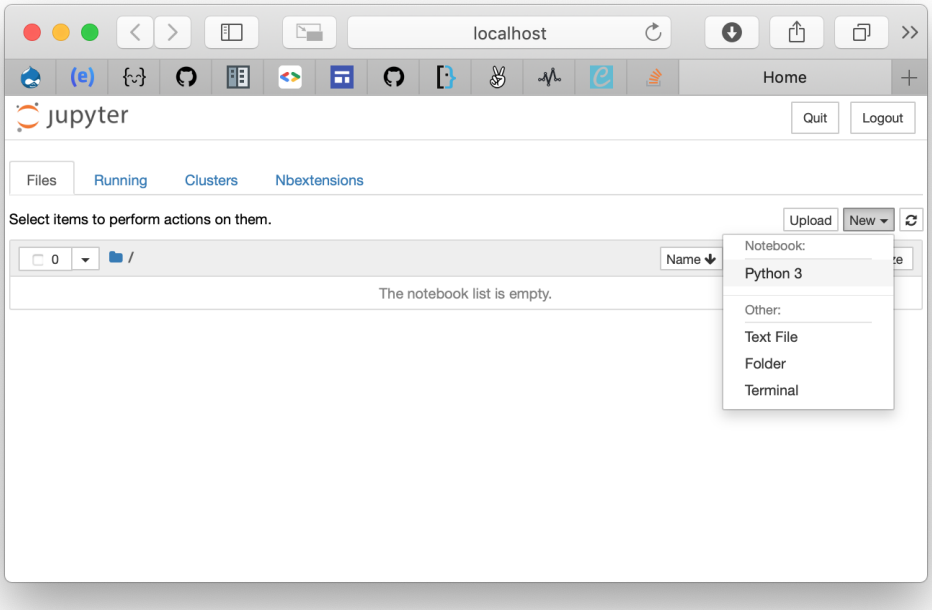

Figure 2: Creating a new notebook.

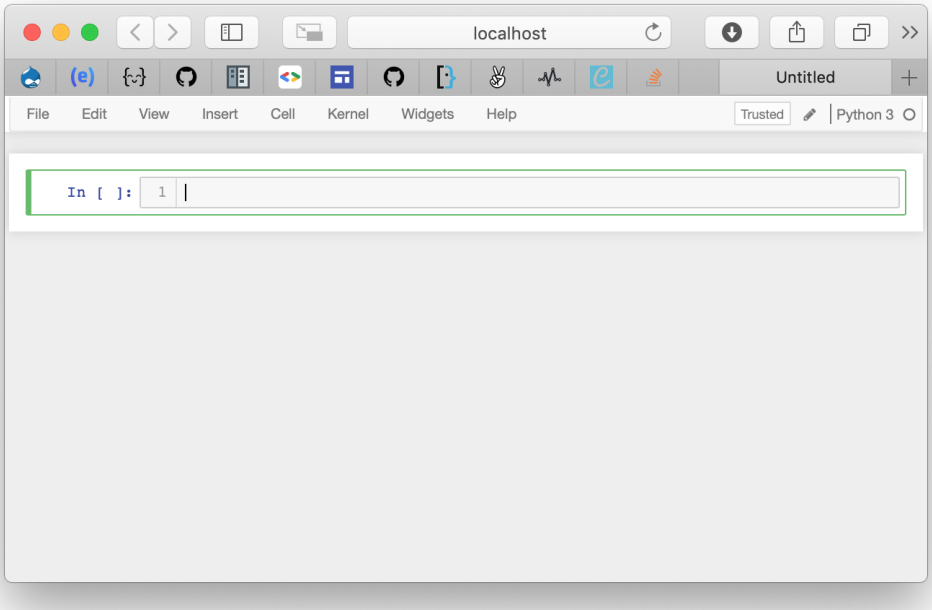

Figure 3: A new jupyter notebook.

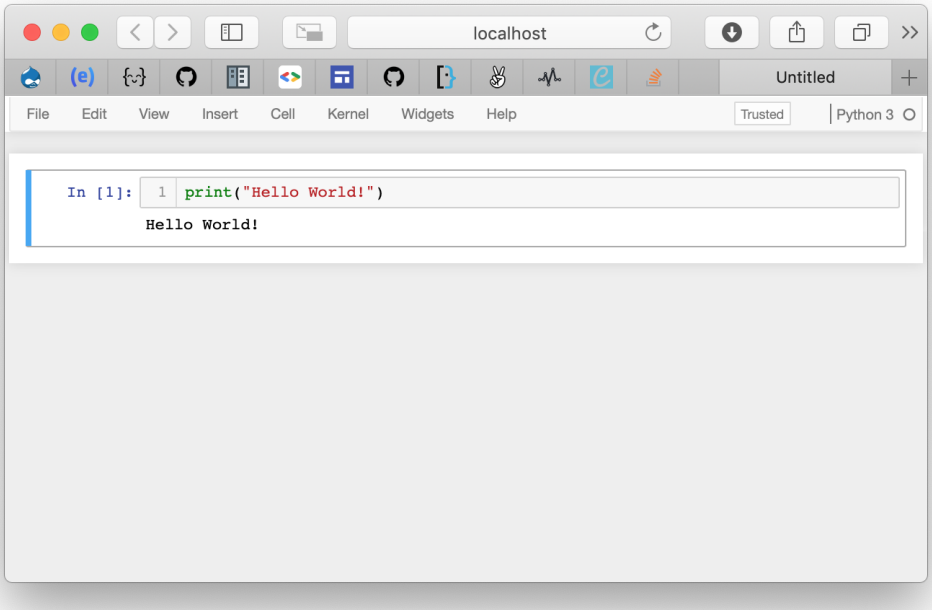

Figure 4: Running Python code in a Code Cell.

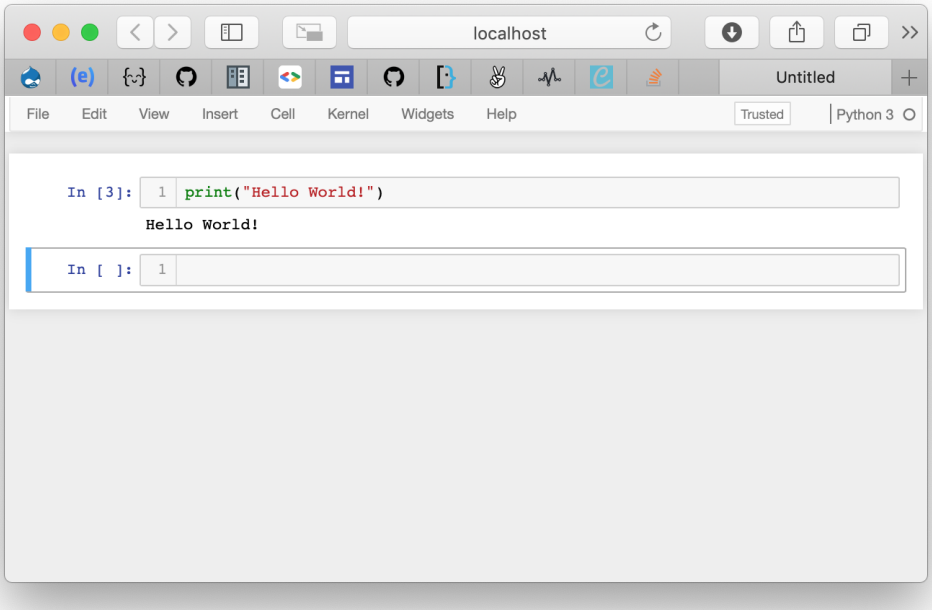

Figure 5: Code Cell in Command Mode.

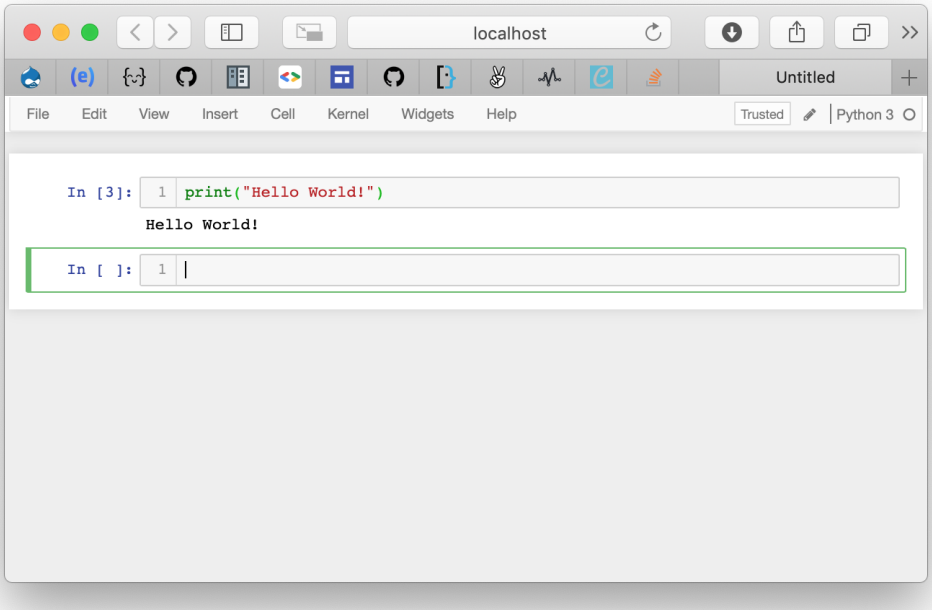

Figure 6: Code Cell in Edit Mode.

markdown cell is a cell that holds rich text, supports the Markdown markup language, and also Latex. We can use these cells to write information relevant to understand the code in code cells. Go into `Edit Mode` and type the following in the markdown cell:

```
A markdown cell holds text.
We can use the Markdown markup language to format the text.
For example:
- **bold text** is created by surrounding text with double asterisks: **.
Markdown cells also support Latex:
- In-line equations: $f(x)=x^2$
- We can also use some environments, like align:
\begin{align}
f(x) &= x^2\\\\
g(x) &= x^3
\end{align}
```

Now, hit `Control-Enter` to execute the markdown cell. This will format the text, taking care of the Markdown formatting and the Latex, and the output will be displayed in the cell (see Figure 7).

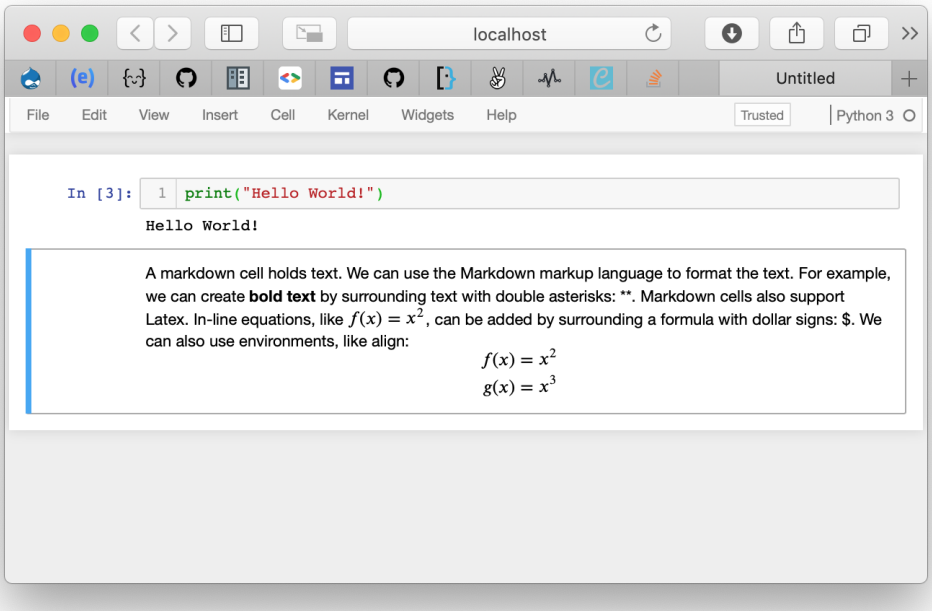

Figure 7: Rendered Markdown Cell.

Notice that after hitting `Control-Enter` you exit `Edit Mode` and go into `Command Mode`. You can edit the markdown cell by hitting `Enter` (going into `Edit Mode` again). You can also hit `Shift-Enter` to format the text and create a new cell below it.

In `Command Mode` you can move between cells using the arrow keys up and down, or with the letters `j` (move down) and `k` (move up)[1]. While in `Command Mode`, you can

[1]Using the letters `k` and `j` as shortcuts to move up and down come from the shortcuts of the text editor `Vim`. In `Vim` you move around text using the letters `h` (left), `j` (down), `k` (up) and `l` (right). This is a more efficient method of moving around text instead of using the arrow keys.

change a cell into a code cell with `y`, and into a markdown cell with `m`. You can create a new cell above the current one with `a`, below the current one with `b`. You can delete a cell with `d d`. You can cut a cell with `x`, and paste it with `v`. When using `v` to paste, the cell will be pasted below the current cell, but you can use `Shift-v` to paste the cell above. You can copy a cell with `c`. We can copy (or cut) multiple cells by first selecting many cells while holding `Shift` (for example, press `Shift-k` a few times), and then pressing either `c` or `x`.

In `Command Mode` you can toggle the line numbers with `l`. If a cell has an output (like in a code cell than you just executed), you can toggle the visibility of the output with `o`. To save the notebook, go to `Command Mode` and use the shortcut `Command-S`, or the menu `File` and button `Save as...` or `Save and Checkpoint`.

In `Edit Mode` on a code cell, you can create new lines by hitting `Enter` or `Control-o`. You can move to the end of a line with `Control-e`, and to the beginning of a line with `Control-a`. Notice that `Control-o` will create a new line below the current line if your cursor is at the end of the line. Also, `Control-o` will create a new line above the current line if your cursor is at the beginning of the line. Lastly, `Control-o` will split the line if your cursor is in the middle of the text in a line.

In `Edit Mode`, you can move around text using the arrow keys. Alternatively, you can use `Control-f` to move forward one letter on a line, `Control-b` to move backwards one letter on a line, `Control-n` to jump to the next line, and `Control-p` to jump to the previous line[2]. To delete a single letter after the cursor use `Control-d`. You can use `Control-k` to delete text from the cursor to the end of the line. To undo edits use `Command-Z`, and to redo edits use `Command-Shift-Z`.

In `Edit Mode`, you can indent the code with `TAB`. Alternatively, you can use `Command-]` to indent the code, and `Command-[` to dedent. You can add a comment line with `Command-/`.

A helpful suggestion is to change the function of the `CAPS LOCK` key on your keyboard. To make commands more natural, change the `CAPS LOCK` to `Control`. This will allow you to quickly maneuver in `Edit Mode` using the keyboard shortcuts (and you will be one step closer to learning how to navigate on Emacs).

If you are on an Windows, the shortcuts might be slightly different due to the lack of a `Command` key. You can always check the shortcuts for your system by going on the menu `Help` and clicking on the button `Keyboard Shortcuts`.

Now you know the basics of how to operate a jupyter notebook. These shortcuts will only get natural if you use them. It is natural to be slow at the beginning, but you should get quicker as you practice more and more. Jupyter notebook is used in the industry and in research labs to share code and results among peers.

Summary

By the end of this lecture notes you should know:

- How to install Python with the installer

[2]Using the shortcuts `Control-b`, `Control-f`, `Control-n` and `Control-p` to move the cursor around text comes from the shortcuts used on the text editor Emacs. This is a more efficient method of moving around text instead of using the arrow keys. In Emacs, this is also more natural since almost all commands rely on the `Control` key.

- What are the main IDEs available for Python
- How programs are found by the terminal and what the `PATH` variable represents
- What is the idea behind a virtual environment
- How to create a virtual environment with `virtualenv`
- How to activate and deactivate the virtual environment
- How to install packages with `pip`
- How to launch Jupyter notebook
- How to create code cells, markdown cells and many shortcuts for the Command Mode and Edit Mode

If you find any typos or issues with these notes, please send me an email with your feedback.

References

- Installing Python
- Managing Python Versions
- Jupyter Notebook for Python
- Virtualenv
- Jupyter Notebook
- Tracking Packages