Python, IPython, Jupyter

Pritam Dalal

April 2, 2019

Python is Interpreted

- Python is an interpreted language. (So are R and Matlab.)
- When you installed Anaconda, you installed a program called the *interpreter*.
- The interpreter sits on you computer and waits for Python commands. Upon receiving one, immediately performs computations encoded in that command.
- Thus far, our experience of issuing commands to the interpreter has been to type code into the cell of a Juypter Notebook, and then press shift + enter.
- Let's explore some other ways to send commands to the interpreter.

Running Python from the Shell (Part 1 of 3)

1. Open up the shell on your machine:
   - Mac OSX - Terminal
   - Windows - ??

2. At the prompt, type `python` and then press -enter-.

3. You should see some text get printed and then following prompt: >>>.

4. You are now in the Python interactive shell.

5. You can type Python commands at the prompt, similar to a Jupyter notebook code cell.

Running Python from the Shell (Part 2 of 3)

Let's try a few commands:

6. `>>> import pandas_datareader as pdr`
7. `>>> df_spy = pdr.get_data_yahoo('SPY')`
8. `>>> df_spy.head()`
9. `>>> df_spy['Adj Close'].plot()`
10. `>>> quit()`

Running Python from the Shell (Part 3 of 3)

- Obviously, this interface leaves a lot to be desired, especially for interactive scientific computing.
- For example, plot most likely won't work on your machine (but there is a way to get them to open in a separate window).
- This was the motivation for IPython.
- IPython is an enhanced wrapper around Python, one that is more suitable for scientific computing.

Running IPyton from the Shell

1. `$ ipython`
2. You will see a few lines of text printed and then: `In: [1]`
3. `In: [2] dir()`
   - notice that `df_spy` is gone from the session
4. `In: [3] import pandas_datareader as pdr`
5. `In: [4] df_spy = pdr.get_data_yahoo('SPY')`
6. `In: [5] df_spy['Adj Close'].plot()` **still didn't work**
7. `In: [6] quit()`

IPython Qt Console

1. The next step in the evolution of IPython was the Qt console, which has a more modern feel and allows for in-line graphics.
2. `$ jupyter qtconsole`
3. `In: [1] import pandas_datareader as pdr`
4. `In: [2] df_spy = pdr.get_data_yahoo('SPY')`
5. `In: [3] %matplotlib inline`
6. `In: [4] df_spy['Adj Close'].plot()` **Success!**
7. `In: [5] quit()`

History (Part 1 of 3)

- Both spreadsheets and computational notebooks started being developed and released in the 1980s.
- Early spreadsheets were VisiCalc, Lotus 1-2-3, and Excel.
- Early notebooks were Maple and Mathematica.
- Python was created in 1990 Guido van Rossum.
- The IPython project was started in 2001 by Fernando Perez.
- IPython began as an enhanced interactive wrapper to Python, designed to facilitate scientific computing.

History (Part 2 of 3)

- IPython was heavily influence by the early computational notebooks, and in the mid-2000s started developing it's own.
- After several stalled attempts, in 2011, the first version of IPython Notebook was developed
- By 2014, the IPython project had come to encompass the interactive shell, the Qt Console, the notebooks, and several other projects - all in a single repository.
  - That's a lot of code.
- Moreover, IPython Notebooks began supporting other languages: first Julia, then R, and now many others. (But Julia, Python, and R are the main three.)

History (Part 3 of 3)

- In 2014, all the language agnostic parts of the IPython project were spun-off and rebranded as Project Jupyter.
- Due to the shared history of Jupyter and IPython, the two often get conflated.
- Today, precise use of the term IPython only has two meanings:
  - the interactive IPython shell
  - the Python backend (kernel) of Jupyter Notebook
- Jupyter has many projects under it's umbrella:
  - Notebooks (you're familiar with)
  - JupyterHub (shared notebooks + more)
  - **JupyterLab** (an IDE for interactive computing projects)

Ways to Execute a Script in JupyterLab

1. Navigate to `packages/01_hello_world` in the Tutorials and Exercises folder.
2. Let's examine the contents of `first_script.py`.
3. You can run this script from a Jupyter notebook:
   - launch the `for_running_script.ipynb` and type along.
   - use the `%run` magic
4. You can also run this script from an IPython console.
   - use the `%run` magic
5. You can also run this script from the shell - brave souls can follow along.

Observations About JupytyerLab

1. The left sidebar give a view to file structure your computer, which allows for easy access and organization of files.
2. An integrated environment for writing and running python code in various ways: scripts, notebook, consoles.
3. At various stages in a large data analysis project, you may need to interact with Python code in these different ways.
4. This makes JupyterLab an ideal IDE for data analysis and scientific computing (very much inline with with vision of Project Jupyter).

Notebook Workflow

- Thus far, our data analysis workflow has been as follows:
  1. read data into a notebook
  2. use functions from preexisting packages to perform calculations on the data
  3. generate calculations and visualizations
  4. write words summarizing findings
- This type of workflow is highly productive for financial data analysis and scientific computing in general.
- This is why Jupyter Notebooks exist.
- Often, you won't need anything else.

When You May Need Something Else

- As your data projects grow larger, you may need to utilize ways of programming other than the interactive notebook style.
- This need usually arises for two reasons:
  1. involved data wrangling
  2. custom functions that you want to use in multiple projects
- For the purposes of data wrangling, I create a lot of scripts and modules. I sometime formalize them into packages.
  - this type of coding is less interactive
  - feels more like programming than analysis

Components of A Large Analysis Project

1. **Motivating Question:** should we take some business action?
2. **Raw Data:** database containing (raw) historical options data.
3. **Third Party Packages:** foundational tools for data wrangling and visualization (`pandas`, `seaborn`, `statsmodel`, `py_vollib`).
4. **Custom Packages:** created by me, consisting of:
   - functions for pulling data from database
   - wrangling functions that I use in many different projects
5. **Wrangling Scripts:** Multiple scripts that perform various wrangling calculations on raw data and produce processed data.

Components of A Large Analysis Project

6. **Processed Data:** clean data files ready for analysis.
7. **Analysis Notebooks:** analysis performed on process data consisting of code, words, calculations.
8. **Decision:** on business action.

Note: In most classroom data analysis settings, including this one, the focus is on #6, #7. Non-interactive programming is the domain of #4, #5.

Let's walk through an example project of mine. It's actually written in R, but the components are the same.

Modular Programming (1 of 2)

- As you can see from my analysis project, there is a lot of code for one particular project.
- In theory, I could have put all the into one single Rmarkdown notebook - but that would have been difficult to understand, let alone maintain.
- This is where *modular* programming comes in handy.
- Modular programming refers to the idea of taking a large coding task and breaking into smaller pieces.
- Programming languages like R and Python facilitate modularity more than Excel.

Modular Programming (2 of 2)

- An essential part of modular programming is spreading code over multiple text files.
- So, at a minimum, the IDE you are programming in should have a convenient way of organizing text files.
  - this is ultimately done with the file system on your OS
  - your IDE should have a convenient view of this
- Python has three coding constructs for facilitating modular programming.
  - functions
  - modules
  - packages

Functions, Modules, Packages

- A user-defined *function* is a block of code that is tied to a name using using the syntax `def function_name(arguments):`
  - it can return a value, or do some other unit of work
- A *module* is a `.py` file that contains valid lines of Python code.
  - That's it!
- A *package* is a folder that contains modules.
  - That's it!
  - no `__init__.py` required

Note: We will use the word *script* more informally to refer to any `.py` that has a set of instructions or computations.

An Example Package

- Let's walk through the example package in `package/02_backtest_py_dev`.
- You won't be able to run the code on your machine, but it will give you a sense for what a simple package looks like.
- As mentioned before, most of my large analysis projects start with historical options data that is stored in a database on my computer.
- In order to get the data from the database into a `DataFrame`, I need to create a set of helper functions.
- This package consists of several of these functions.

Creating a Simple Package

1. Let's create a simple package together.
2. In JupyterLab navigate to `package/03_toy_wrangler`.
3. Type along with me.