Chapter 2: Building Your Analytics Toolbox: A Primer on Using R & Python For Security Analysis

If you add a little to a little and do this often, soon the little will become great.

Hesiod

Before we jump right into the various use cases in the book it’s important to ensure you at least have a basic familiarity with the two most prominent languages featured in nearly all of the scenarios: Python (http://www.python.org/) and R (http://www.r-project.org/). While there are an abundance of tools available for data analysis, we feel these two provide virtually all the features necessary to help you go from data to discovery with the least amount impedance.

A sub-theme throughout the book, and the distilled process at the heart of security data science, is: **idea**, **exploration**, **trial** (and *error*) and **iteration**. It is ineffective at best to attempt to shoehorn this process into a traditional *edit*/*compile*/*run* workflow found in most traditional languages and development environments. The acts of performing data analysis tasks and creating informative visualizations are highly interactive and iterative endeavors. Despite all of their positive features, even standalone Python and R do not truly enable rich, dynamic interaction with code and data; but, when coupled with IPython (http://ipython.org/) and RStudio (http://www.rstudio.com/), respectively, they are both transformed into powerful exploration tools, enabling rapid development and testing of everything from gnarly data munging to generating sophisticated visualizations.

This chapter will: provide pointers to installation resources for each tool, introduce core features of each language and development environment and explain the structure of the examples you will find in the remaining chapters of the book.

Why Python; Why R; *And*, Why Both?

A discussion of which programming language is better than another for a certain set of tasks often turns (quickly) into a religious war of words that rarely wins converts and never becomes fully resolved. As a security data scientist, you will find that you do not have the luxury of language bias since there will be times that one language shines in one area while a different one shines in another, and you will need the skills of a diplomat to bring them both together to solve real problems.

We’ve honed in on both R/RStudio and Python/IPython/pandas in this book, as they are the two leading data analysis languages/environments with broad similarities but also with unique elements that make them work well for some tasks and not others. As we go into the rationale behind each choice and as you become proficient in one or both environments, do not lull yourself into a sense of complacency. A hallmark of a good data scientist is adaptability and you should be continually scouring the digital landscape for emerging tools that will help you solve problems. We’ll introduce you to some of these upstarts in Chapter 13.

Why Python?

Guido van Rossum created the Python programming language in December of 1989 to solve a problem. He and his colleagues needed a common way to orchestrate system administration tasks that could take advantage of specific features in the operating systems they were using at that time. While there were existing interpreted, administrator-friendly tools and languages availabl, none were designed (from Guido van Rossum’s point of view) with either the flexibility or extensibility features baked into the design principles of Python.

Pyhon’s flexibility and extensibility (and the fact that it was free as in both “speech” and “beer”) were especially appealing to the scientific, academic and industrial communities starting in the early 2000s, and innovators in these fields quickly adapted this general purpose programming language to their own disciplines to solve problems easier than—ostensibly—the domain-specific languages available at that time.

You would have to search long and hard to find a file-type Python cannot read, a database Python cannot access and an algorithm Python cannot execute. As you familiarize yourself with the language, Python’s ability to acquire, clean and transform source data will quickly amaze you, but those tasks are just the early steps in your analysis and visualization process. It wasn’t until 2008 that the pandas (http://pandas.pydata.org/) module was created by AQR Capital Management to provide “Pythonic” counterparts to the analytical foundations of languages like R, SAS or MATLAB, which is where the “real fun” begins.

While Python’s interpreter provides an interactive execution shell, aficionados recognized the need to extend this basic functionality and developed an even more dynamic and robust interactive environment—IPython—to fill the need. When coupled with the pandas module, any budding data analyst now has a mature and data-centric toolset available to drive their quest for knowledge.

Why R?

Unlike Python, R’s history is inexorably tied to its domain specific predecessors and cousins, as it is 100% focused and built for statistical data analysis and visualization. While it, too, can access and manipulate various file types and databases (and was also designed for flexibility and extensibility), R’s lisp- and S-like syntax and extreme focus on foundational analytics-oriented data types has kept it, mostly, in the hands of the “data crunchers”.

Base R makes it remarkably simple to run fundamental statistical analyses on your data and then generate informative and appealing core visualizations with just a few lines of code. More modern R libraries such as plyr and ggplot2 extend and enhance these base capabilities and are the foundations of many of mind- and eye-catching examples of modern data analysis and visualization you have no doubt come across on the internet.

Like Python, R also provides an interactive execution shell that has enough basic functionality for general needs. Yet, the desire for even more interactivity sparked the development of RStudio, which is a combination of IDE, data exploration tool, and iterative experimentation environment that exponentially enhances R’s default capabilities.

Why Both?

If all you have a hammer, everything starts looking like a nail. There are times when the flexibility of a general purpose programming language will come in very handy and that should definitely be a situation where you use Python. There are other times when three lines of R code will do something that may take thirty or more lines of Python code to accomplish. Since your ultimate goal is to provide insightful and accurate analyses as quickly and visually appealing as possible, knowing which tool to use for what job is a critical insight you must develop to be as effective and efficient as possible.

We would be a bit dishonest, though, if we did not concede that there are some things that Python can do (easily or at all) that R cannot, and vice-versa. We’ll touch upon some of these in the use-cases throughout the book, but many of the—ah—“learning opportunities” will only come from performing your own analyses, getting frustrated (which is the polite way of saying “stuck”) and only finding resolution by jumping to another tool to “get stuff done”. This situation comes up frequently enough that there is even a “rJython” package for R that lets you call Python code from R and “rpy” and “rpy2” modules for Python that let you call R code from Python.

By having both tools in your toolbox, you should be able to tackle most, if not all, of the tasks that come your way. Both R and Python have vibrant communities that are eager to provide assistance and even help in the development of new functions or modules to fit emerging needs.

Jumpstarting Your Python Analytics With Anaconda

It *is* possible to setup an effective and efficient installation of Python, IPython and pandas from the links we’ve provided, especially if you are already familiar or proficient with Python. However, even the most stalwart Python aficionado can find it challenging to manage dependencies and updates for the numerous necessary components. This is especially true if you have to manage analytics processes across multiple operating systems and environments.

To facilitate both ease of installation and maintenance, we highly recommend using the freely available Anaconda distribution from Continuum Analytics (https://store.continuum.io/). Anaconda works on Linux, Microsoft Windows and Mac OS X; supports Python 2.6, 2.7 and 3.3; and, incorporates a meta-package manager that will help you keep current with changes in every dependent package and module. Anaconda also makes it very straightforward to jump between Python versions without trashing your environment. For those working in organizations that shy away from open source solutions, Continuum Analytics also offers commercially supported options for Anaconda.

Given that there is a comprehensive installation, setup and update guide available from Continuum Analytics (<http://docs.continuum.io/anaconda/install.html>), we will not go over step-by-step instructions on how to install Anacondas for each platform. After you perform the base installation and have your environment variables setup correctly, getting started should be as easy as running the following at a command prompt (regardless of platform):

**ipython notebook --pylab=inline**

[NotebookApp] Using existing profile dir:

u'/Users/hrbrmstr/.ipython/profile\_default'

[NotebookApp] Serving notebooks from /Users/hrbrmstr

[NotebookApp] The IPython Notebook is running at:

http://127.0.0.1:8888/

[NotebookApp] Use Control-C to stop this server and

shut down all kernels.

That command tells IPython to startup in “HTML notebook mode” and to display all graphical output inline (i.e. in the web browser with the Python code) versus in a separate window. Your “profile dir” will (hopefully) be different but the overall shell output should look extremely similar. To see if everything works, point your favorite browser to the listed URL (which should be <http://127.0.0.1:8888/>) and ensure you see a similar output to Figure 2.1.

Figure 2.1 IPython Notebook Start Page [793725c02f001.png]

From that point, create a new notebook (via the “*New Notebook*” button) and you should be in an interactive environment similar to Figure 2.2.

Figure 2.2 IPython Notebook Interactive Environment [793725c02f002.png]

Now that you are connected to the IPython Notebook server this would be an excellent time to run through the following tests from the set of startup examples on the IPython site (https://github.com/ipython/ipython/tree/master/examples/notebooks#a-collection-of-notebooks-for-using-ipython-effectively):

* Part 1 – Running Code
* Part 2 – Basic Output
* Part 3 – Pylab and Matplotlib
* Part 4 – Markdown Cells
* Part 5 – Rich Display System

to validate that your environment is setup correctly.

type="note"

Managing Your Python Environment

When using Anaconda, it’s important to manage all module and core Python updates through the conda update manager to ensure consistency. Failure to do so may result in a “wonky” environment and may force you to do a complete un-install and re-install of the Anaconda package.

Unlike RStudio, IPython and IPython Notebooks do not provide a full-fledged IDE environment for managing code. There are analytics-oriented IDE tools for Python, such as Spyder (<https://code.google.com/p/spyderlib/>), which comes with Anaconda and has similar features to RStudio and other IDEs, but we recommend using a plain, old text editor that has Python syntax recognition and highlighting/indenting capabilities to augment your work in browser-based IPython Notebooks. There are even some editors that have extensions that enable execution of in-editor code and code snippets directly in IPython Notebooks. For now, stick with the basics until you become more comfortable with the iterative execution and exploration environment IPython has to offer.

Understanding The Python Data Analysis And Visualization Ecosystem

While there are scores of modules available for Python, a few stand out when it comes to crunching data. Here are some that you will find yourself using in nearly every project:

* *NumPy* (<http://www.numpy.org/>) – a library providing foundational capabilities for creating multi-dimensional containers of generic data, performing a wide range of operations on data and generating random numbers. It also implements the capability to “broadcast” operations to Python objects, which can make for succinct and highly efficient code;
* *SciPy library* (<http://www.scipy.org/scipylib/index.html>) – built on top of NumPy, this library makes quick work of array-oriented operations and provides a facility to expand NumPy’s “broadcast” operations to other types of data elements in Python; it also provides additional statistical operations;
* *Matplotlib* (<http://matplotlib.org/>) – the de-facto way to turn your data into production-quality images in Python;
* *Pandas* – a library providing high-performance, easy-to-use data structures and data analysis tools; pandas introduces the Data.Frame type into the Python namespace which we will discuss in more detail later in this chapter; while this may cause some die-hard Python folks to cringe, pandas—in essence—makes Python more like R and should make it easier for you to jump between languages.

These modules, combined with IPython, are sometimes referred to the core components of the “*SciPy stack*” (which is confusing, since it contains the “*SciPy library*”), and you can read more about the stack at <http://www.scipy.org/>.

As you make your way through this ecosystem, you will notice the following pattern emerge:

import numpy as np

import scipy as sp

import matplotlib as mpl

import matplotlib.pyplot as plt

import pandas as pd

The import statement brings the namespace and functionality of a particular library/module into the current Python working session and the “as” component of the statement provides an abbreviated reference for the functions, objects and variables in the module. Adding a way to quickly generate those imports would be a wise addition to your editor of choice.

type="note"

Python “Gotchas”

There are two features of Python that are liable to both frustrate and perhaps become problematic for new users. The first “gotcha” is whitespace. Spaces are significant in Python code. There are no “{}” braces or “begin/end” pairs to signify a block of code. You must use consistent indentation to identify groups of statements that will execute together. Inconsistency will result either in error messages from the interpreter or your code failing to work as expected. Most modern text editors or IDE can be configured to take care of this for you.

The second “gotcha” is the lack of a requirement to declare variables before using them. Initializing a variable named “breaches” to some value then inadvertently referring to it later as “breached” may not throw an error in the interpreter, but will most assuredly generate unexpected output.

You will, of course, use other packages for connecting to databases, reading from files and performing other functions and you can burn countless hours perusing all the available modules at the Python Package Index (PyPI), https://pypi.python.org/pypi, but the ones associated with the *SciPy stack* will become familiar and regular companions on your data science journey.

Setting Up Your R Environment