[[CE by Kezia Endsley]]

[[TE by Russell Thomas – color code: GREEN checks OK, RED has error that needs to be fixed, YELLOW needs attention. Some yellow is by CE]]

Chapter 2: Building Your Analytics Toolbox: A Primer on Using R and Python for Security Analysis

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

⎯Hesiod

Before you 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/). Although 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 the *edit*/*compile*/*run* workflow found in most traditional languages and development environments. The acts of performing data analyses 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. However, when they are coupled with IPython (http://ipython.org/) and RStudio (http://www.rstudio.com/), respectively, they are transformed into powerful exploration tools, enabling rapid development and testing of everything from gnarly data munging to generating sophisticated visualizations.

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

[[Authors: Just a few notes about why I made the edits I did in case you’re wondering. We avoid using “we” when you mean author and reader and prefer “you” instead (“we” to represent several authors is fine). We avoid using past tense and future tense when possible and use present tense to discuss any information in the book. Also, we use the serial comma. Again, just FYI in case you’re wondering. Very nicely written chapter. Kezia]]

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. There will be times when one language shines in one area while a different one shines in another, and you 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 read about 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 introduce you to some of these upstarts in Chapter 13.

[[Authors: According to my TOC, there is no chapter 13? Kezia]]

[[Authors: It might be good to add a sentence or two about how long it takes to become proficient in Python and R, given that the reader has some prior experience with programming (C, C++, Java, etc.), scripting (Pearl, shell), statistical software (SAS, SPSS), or computational mathematics (MatLab, Mathematica). For example, how long did it take for *you* to become proficient? What was the learning curve like? What’s the best approach – a little at time or total emersion? ]]

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. Although there were existing interpreted, administrator-friendly tools and languages available, 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.

Python’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. 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 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.

Although 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, budding data analysts now have 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 percent focused and built for statistical data analysis and visualization. Although 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 plus 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 extensive statistical analyses on your data and then generate informative and appealing 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 cutting-edge 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 integrated development environment (IDE), data exploration tool, and iterative experimentation environment that exponentially enhances R’s default capabilities.

Why Both?

If all you have is a hammer, everything starts looking like a nail. There are times when the flexibility of a general-purpose programming language comes in very handy, which is when you use Python. There are other times when three lines of R code will do something that may take 30 or more lines of Python code (even with pandas) to accomplish. Since your ultimate goal is to provide insightful and accurate analyses as quickly and as visually appealing as possible, knowing which tool to use for which 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 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 finding resolution by jumping to another tool to “get stuff done.” This situation comes up frequently enough that there is even an rJython package for R that lets you call Python code from R scripts and rpy and rpy2 modules for Python that let you call R code from Python scripts.

[[Authors: We don’t use quotes around terms that have code font applied, unless the quotes are part of the actual code. Kezia]]

By having both tools in your toolbox, you should be able to tackle most, if not all, of the tasks that come your way. If you do find yourself in a situation where you need functionality you don’t have, 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 Canopy

It *is* possible to set up an effective and efficient installation of Python, IPython, and pandas from the links we’ve listed above, especially if you are already familiar or proficient with Python, however we don’t recommend it

[[Authors: I suggest adding a sentence that describes what installing Python means – i.e. it’s the core language/interpreter plus dozens or scores of libraries. The point is that it is not like installing a C compiler or an integrated product like *SAS* or *Mathematica*.]]

Even the most stalwart Python aficionado can find it challenging to manage dependencies and updates for the numerous necessary components. This can waste hours of your time. 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 Enthought Canopy Python data analysis environment (<https://www.enthought.com/products/canopy/>). Canopy works on Linux, Microsoft Windows, and Mac OS X; has a built-in Python integrated development environment (IDE); incorporates a meta-package manager that will help you keep current with changes in every dependent package and module; and also comes with an IPython console. For those working in organizations that shy away from open source solutions, Enthought also offers commercially supported options for Canopy.

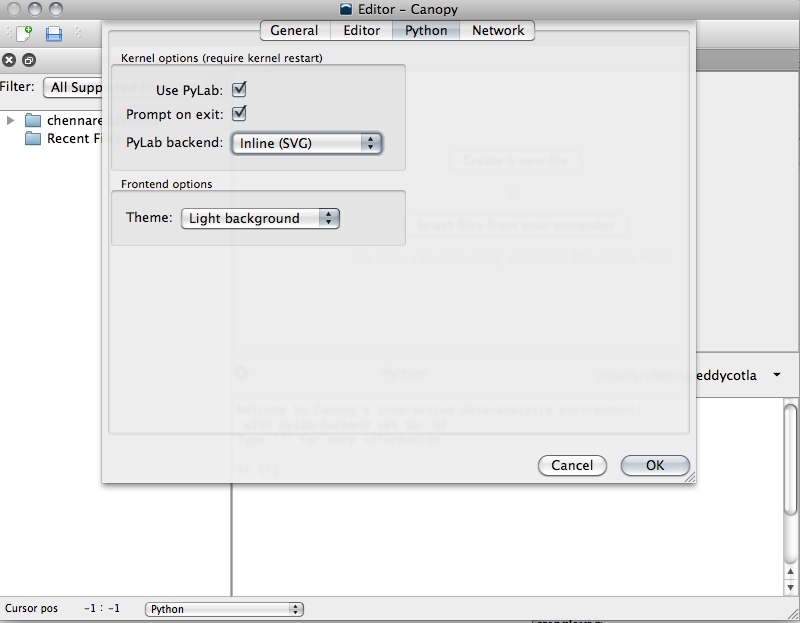
Given that there is a comprehensive installation, setup, and update guide available (http://docs.enthought.com/canopy/quick-start.html), we will not go over step-by-step instructions on how to install Canopy for each platform, but we strongly recommend reviewing the documentation before attempting any of the Python examples in the book. Once the base installation is complete, getting started should be as straightforward as opening up the Canopy application, which will display the welcome screen (see Figure 2.1).

Figure 2.1 Canopy Welcome Screen [793725c02f001.png]

One of first steps you should perform is to instruct Canopy to display all images *inline* within the IPython console. This is an optional step, but it will help keep all output self contained within the Canopy environment. You can change this setting once you have an open Canopy editor session by going into the Preferences window, finding the Python tab and selecting the Inline (SVG) option for the PyLab Backend preference (see Figure 2.2).

Figure 2.2 Canopy IDE with Preferences Open [793725c02f002.png]

Authors: This figure shows the Canopy console but not the Preferences window, as referenced in the text. This is confusing. It would be better to have a figure like this:



To validate that your environment is set up properly, run the following code in the IPython console area in the editor

import pandas as pd

import numpy as np

np.random.seed(1492)

test\_df = pd.DataFrame({ "var1": np.random.randn(5000) })

test\_df.hist()

and verify that it produces the output seen in Figure 2.3. If it does, you have the basic environment installed and are ready to start working through the data analysis examples. If the bar chart is not displayed, you may need to check your installation steps or verify you have the proper graphics display options mentioned earlier.

Figure 2.3 Test IPython Console Output [793725c02f003.png]

Once everything is working properly, you should carve out 10 minutes to read through “Learn Python in 10 Minutes” (<http://www.stavros.io/tutorials/python/>) by Stavros Korokithakis if you are not familiar with Python and then spend 10 additional minutes to go through the “10 Minutes to Pandas” tutorial (<http://pandas.pydata.org/pandas-docs/dev/10min.html>) to learn a bit more about the pandas data analysis module.

Understanding the Python Data Analysis And Visualization Ecosystem

Although there are scores of libraries available for Python, a few stand out when it comes to crunching data.

[[Authors: I suggest adding:

We call it an ‘ecosystem’ because each library is developed and supported by a different organization or community. They coordinate with each other, but the coordination is loose. ]]

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 most powerful and commonly used library 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. Although 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 code 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.

[[Authors: I’m concerned that terms like “namespace” are being introduced without being defined or contextualized. It might be good to create a box titled something like “10 Things You Need to Know About Python” and then use it to define terms like “namespace”. The “Gotchas” box below is very good.]]

Since you’ll be using many of the components of each of the modules in the *SciPy stack* on a regular basis, you will save time and typing if you use a base template file with these and other much reused code built into it.

type="general"

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 in 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 nifty 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.

[[Authors: it might be useful to add a paragraph with advice on how to keep your Python installation up to date using Homebrew or something else.

Also you should probably add a paragraph on the version 2.6/2.7 vs. sersion 3.x issue. This is confusing for many people, myself included, so it would help if you could offer simple guidance.]]

Setting Up Your R Environment

To build your R/RStudio environment you will need to download and install R (<http://cran.rstudio.com/>) then do the same for RStudio (<http://www.rstudio.com/ide/download/>). Both links provide full installation details for Linux, Windows, and Mac OS X systems so we won’t delve into the minutiae in this section. You do, however, need to make a choice when you install RStudio as it comes in two flavors: Desktop and Server. Both provide the same core features:

* Built-in IDE
* Data structure and workspace exploration tools
* Quick access to the R console
* R help viewer
* Graphics panel viewer
* File system explorer
* Package manager
* Integration with version control systems

The primary difference is that one runs as a standalone, single-user application (RStudio Desktop) and the other (RStudio Server) is installed on a server, accessed via browser, and enables multiple users to take advantage of the compute infrastructure. If you are not familiar with R or RStudio, begin by downloading and installing RStudio Desktop (all examples in this book involving RStudio assume you are working in the Desktop version).

For those of you limited to working with commercially supported tools, Revolution Analytics (<http://www.revolutionanalytics.com/support/>) provides commercial offerings and technical support for R.

Once everything is installed, open RStudio and verify you see the default workspace, which should look similar to Figure 2.4.

Figure 2.4 RStudio Default Workspace [793725c02f004.png]

If all is working correctly, you should take some time to walk through “A (Very) Short Introduction to R” by Paul Torfs and Claudia Brauer (<http://cran.r-project.org/doc/contrib/Torfs%2BBrauer-Short-R-Intro.pdf>). It will run through just enough of the basics of the R language and RStudio environment to make you dangerous.

Although you can use the built-in package manager with RStudio to install packages, you will eventually come to the realization that using the console method is much more convenient. To get familiar with this process right away, you should install the ggplot2 package, which is the primary graphics library used in the book’s examples. Installation is as straightforward as entering the following into the RStudio console pane:

[[Authors: I changed “curly” quotes below in code to straight quotes. Although they might not look curly per se, they are converted to curly quotes in layout and can cause issues when/if book code is online or in ebook format. Kezia]]

**> install.packages("ggplot2")**

Installing package(s) into '/Library/Frameworks/R.framework/Versions/2.15/Resources/library'

(as 'lib' is unspecified)

trying URL 'http://cran.mirrors.hoobly.com/bin/macosx/leopard/contrib/2.15/ggplot2\_0.9.3.1.tgz'

Content type 'application/x-gzip' length 2659920 bytes (2.5 Mb)

opened URL

==================================================

downloaded 2.5 Mb

The downloaded binary packages are in

/var/folders/qg/vmtfcv1j7vjfq\_p5zw86mk7mxkhymk/T//RtmpiZ5FD3/downloaded\_packages

[[Kevin: CodeScreen format correct above? Kezia]]

Run the following code to verify that ggplot2 has been installed correctly and to ensure your R/RStudio environment is functional:

library(ggplot2)

set.seed(1492)

test.df = data.frame(var1=rnorm(5000))

ggplot(data=test.df) + geom\_histogram(aes(x=var1))

If there are no errors and you see the bar chart in Figure 2.5, your environment is ready to run through the examples in the book.

Figure 2.5 Test R/RStudio Output [793725c02f005.eps]

Like Python, R has a vast repository of useful modules that can simplify many tasks. You will be introduced to a few of them in the coming chapters but should also peruse the Comprehensive R Archive Network (CRAN) (<http://cran.r-project.org/web/packages/>) to see the breadth and depth covered by a host of contributors.

Introducing Data Frames

If you are coming from another programming language you should have a basic understanding of general data types such as strings, integers, and arrays. R and Python offer the standard set of data types, but both have one data type in common—the *data frame*—which truly gives them power. On the surface, a data frame is just a way to hold tabular data (the type of data you see organized in a typical Excel spreadsheet) and may feel like a two-dimensional (2D) array. If you dig a bit deeper, though, you will find that these data frames are really an all-in-one combination of a database table, matrix, 2D array, and pivot table with many additional time-saving features.

Much like a database table, each column in a data frame has a column name and holds elements of the same *type* of data. You can perform operations on whole columns, rows, or subsets of each. Adding, merging, flattening, expanding, changing, deleting, and searching for data are all—usually—one-line operations in both languages as are methods to read and write the contents of data frames to and from files.

[[Authors: You might add this explanation:

“In essence, Python and R achieve this expressive power by putting intelligence into the data structure and the functions that operate on them. In contrast, other programming languages have less sophisticated data structures, meaning you need to write your own code and create your own data structures to achieve similar results.”

The following code provides a compact overview of data frame operations on both R and Python, but it is still highly recommended that you check out the aforementioned introductory resources before moving into Chapter 3, “Learning the “Hello World” of Security Data Analysis.” As indicated in the Introduction, you can find all code on the book’s companion website.

R Dataframe Example

# create a new data frame of hosts & high vuln counts

**assets.df <- data.frame(**

**name=c("danube","gander","ganges","mekong","orinoco"),**

**os=c("W2K8","RHEL5","W2K8","RHEL5","RHEL5"),**

**highvulns=c(1,0,2,0,0))**

# take a look at the data frame structure & contents

**str(assets.df)**

'data.frame': 5 obs. of 3 variables:

$ name : Factor w/ 5 levels "danube","gander",..: 1 2 3 4 5

$ os : Factor w/ 2 levels "RHEL5","W2K8": 2 1 2 1 1

$ highvulns: num 1 0 2 0 0

**head(assets.df)**

name os highvulns

1 danube W2K8 1

2 gander RHEL5 0

3 ganges W2K8 2

4 mekong RHEL5 0

5 orinoco RHEL5 0

# show a "slice" just the operating systmes

# by default R creates "factors" for categorical data so

# we use as.character() to expand the factors out

**head(assets.df$os)**

[1] W2K8 RHEL5 W2K8 RHEL5 RHEL5

Levels: RHEL5 W2K8

# add a new column

**assets.df$ip <- c("192.168.1.5","10.2.7.5","192.168.1.7",**

**"10.2.7.6", "10.2.7.7")**

# extract only nodes with more than one high vulnerabilty

**head(assets.df[assets.df$highvulns>1,])**

name os highvulns ip

3 ganges W2K8 2 192.168.1.7

# create a 'zones' column based on prefix IP value

**assets.df$zones <-**

**ifelse(grepl("^192",assets.df$ip),"Zone1","Zone2")**

# take a final look at the dataframe

**head(assets.df)**

name os highvulns ip zones

1 danube W2K8 1 192.168.1.5 Zone1

2 gander RHEL5 0 10.2.7.5 Zone2

3 ganges W2K8 2 192.168.1.7 Zone1

4 mekong RHEL5 0 10.2.7.6 Zone2

5 orinoco RHEL5 0 10.2.7.7 Zone2

Python (pandas) DataFrame Example

**import numpy as np**

**import pandas as pd**

# create a new data frame of hosts & high vuln counts

**assets\_df = pd.DataFrame( {**

**"name" : ["danube","gander","ganges","mekong","orinoco" ],**

**"os" : [ "W2K8","RHEL5","W2K8","RHEL5","RHEL5" ],**

**"highvulns" : [ 1,0,2,0,0 ]**

**} )**

# take a look at the data frame structure & contents

**print(assets\_df.dtypes)**

highvulns int64

name object

os object

dtype: object

**assets\_df.head()**

highvulns name os

0 1 danube W2K8

1 0 gander RHEL5

2 2 ganges W2K8

3 0 mekong RHEL5

4 0 orinoco RHEL5

# show a "slice" just the operating systmes

**assets\_df.os.head()**

0 W2K8

1 RHEL5

2 W2K8

3 RHEL5

4 RHEL5

Name: os, dtype: object

# add a new column

**assets\_df['ip'] = [ "192.168.1.5","10.2.7.5","192.168.1.7",**

**"10.2.7.6", "10.2.7.7" ]**

# show only nodes with more than one high vulnerabilty

**assets\_df[assets\_df.highvulns>1].head()**

highvulns name os ip

2 2 ganges W2K8 192.168.1.7

# divide nodes into network 'zones' based on IP address

**assets\_df['zones'] = np.where(**

**assets\_df.ip.str.startswith("192"), "Zone1", "Zone2")**

# get one final view

**assets\_df.head()**

highvulns name os ip zones

0 1 danube W2K8 192.168.1.5 Zone1

1 0 gander RHEL5 10.2.7.5 Zone2

2 2 ganges W2K8 192.168.1.7 Zone1

3 0 mekong RHEL5 10.2.7.6 Zone2

4 0 orinoco RHEL5 10.2.7.7 Zone2

The data frame is the core data structure you will find yourself using in either language for most analytics projects. It lets you focus on *what* you want to do with the data versus *how* to do it. This is one of the core differences between domain-specific and general-purpose programming languages. If you were still on the fence about switching to R or Python for performing data analysis, hopefully this brief introduction to the power of each language has helped convince you of their efficacy.

Organizing Analyses

Finally, as you prepare to jump into data-analysis projects, it’s good idea to set up an area where you organize input data, analysis scripts, output (visualizations, reports, and/or data), and any supporting documentation. For the purposes of the examples in this book, we use the following directory structure:

/book/ch03

|-*R*

|-data

|-docs

|-output

|-python

|-support

|-tmp

Like most elements of programming, there is no single best way to set up this structure, but you should strive to find one that works for you and stick with it. A great way to do that is to take a lesson from modern web framework builders and use a simple setup shell script that builds the structure for you:

Sample Analysis Preparation Script (Bourne Shell Script)

#!/bin/sh

#

# prep: prep analytics directory structure

#

# usage: prep DIRNAME

#

DIR=$1

if [ ! -d "${DIR}" ]; then

mkdir -p ${DIR}/*R* \

${DIR}/data \

${DIR}/docs \

${DIR}/output \

${DIR}/python \

${DIR}/support \

${DIR}/tmp

> ${DIR}/readme.md

ls -lR ${DIR}

else

echo "Directory "${DIR}" already exists"

fi

I was not able to get this script to work on Terminal on a Mac OS X. I’m not very experienced in shell scripts, but I ran “hello.sh” successfully. When I run this script, I get the following:

RTs-MacBook-Pro:~ russ$ sh -x prep.sh /book

'iecho 'Directory already exists

ficho Directory already exists

I searched and there are no directories named “ficho” or “iecho”. I also tried creating the directory “book” ahead of this command, but got the same error. Finally, if I issue the command with no directory parameter, it doesn’t give me the “usage”. Instead, I get the same result as above.

Sample Analysis Preparation Script (Windows Shell Script)

SET PDIR=%1

IF EXIST %%PDIR GOTO HAVEPDIR

MKDIR %%PDIR

MKDIR %%PDIR\R

MKDIR %%PDIR\data

MKDIR %%PDIR\docs

MKDIR %%PDIR\output

MKDIR %%PDIR\python

MKDIR %%PDIR\support

MKDIR %%PDIR\tmp

<NUL (SET/P Z=) >%%PDIR\readme.md

DIR %%PDIR

:HAVEPDIR

ECHO "Directory exists"

I am not able to test this code because I’m on a Mac, not Windows.

You now only need to type prep *NAME* whenver you want to start a new project (so, for this project, you type prep ch03). As you develop your own styles and patterns, you can expand this script to include the generation of various templates and even initialization of source code repositories. Once the structure is in place, it’s time to retrieve, explore, and analyze some data!

[[Authors: Do readers actually type the quotes too when they type prep ch03? I assumed not. If they do type the quotes too, they should also be in code font. Also, shouldn’t that read ch02, NOT ch03? Kezia]]

[[Authors: Should there be a Summary section, like in chapter 3?? This is something that is usually consistent across all chapters of a book. Kezia]]

[[In my opinion, this chapter doesn’t need a summary because the content is procedural. The “summary” is the accomplishment of having Python and R environments installed and running.]]