

Applied Data Science with Python and Jupyter

Use powerful industry-standard tools to unlock new, actionable insights from your data



Packt>

www.packt.com

Alex Galea

Applied Data Science with Python and Jupyter

Use powerful industry-standard tools to unlock new, actionable insights from your data



Packt

www.packt.com

Alex Galea

Contents

1. Preface

1. About the Book

1. About the Author
2. Objectives
3. Audience
4. Approach
5. Minimum Hardware Requirements
6. Software Requirements
7. Installation and Setup
8. Installing Anaconda
9. Updating Jupyter and Installing Dependencies
10. Additional Resources
11. Conventions

2. Jupyter Fundamentals

1. Introduction

2. Basic Functionality and Features

1. What is a Jupyter Notebook and Why is it Useful?
2. Navigating the Platform
3. Exercise 1: Introducing Jupyter Notebooks
4. Jupyter Features
5. Exercise 2: Implementing Jupyter's Most Useful Features
6. Converting a Jupyter Notebook to a Python Script
7. Python Libraries
8. Exercise 3: Importing the External Libraries and Setting Up the Plotting Environment

3. Our First Analysis - The Boston Housing Dataset

1. Loading the Data into Jupyter Using ...

APPLIED DATA SCIENCE WITH PYTHON AND JUPYTER

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Author: Alex Galea

Reviewer: Elie Kawerk

Managing Editor: Mahesh Dhyani

Acquisitions Editor: Aditya Date

Production Editor: Samita Warang

Editorial Board: David Barnes, Ewan Buckingham, Simon Cox, Manasa Kumar, Alex Mazonowicz, Douglas Paterson,

Dominic Pereira, Shiny Poojary, Saman Siddiqui, Erol Staveley, Ankita Thakur, and Mohita Vyas

First Published: October 2018

Production Reference: 2051218

ISBN: 978-1-78995-817-1

Table of Contents

Preface

Jupyter Fundamentals

INTRODUCTION

BASIC FUNCTIONALITY AND FEATURES

WHAT IS A JUPYTER NOTEBOOK AND WHY IS IT USEFUL?

NAVIGATING THE PLATFORM

EXERCISE 1:

INTRODUCING JUPYTER

NOTEBOOKS

JUPYTER FEATURES

EXERCISE 2:

IMPLEMENTING

JUPYTER'S MOST

USEFUL FEATURES

CONVERTING A JUPYTER

NOTEBOOK TO A

PYTHON SCRIPT

PYTHON LIBRARIES

EXERCISE 3: IMPORTING

THE EXTERNAL

LIBRARIES AND SETTING

UP THE PLOTTING

ENVIRONMENT

OUR FIRST ANALYSIS -

THE BOSTON HOUSING DATASET

LOADING THE DATA INTO JUPYTER USING A PANDAS DATAFRAME

EXERCISE 4: LOADING THE BOSTON HOUSING DATASET

DATA EXPLORATION

EXERCISE 5: ANALYZING THE BOSTON HOUSING DATASET

INTRODUCTION TO PREDICTIVE ANALYTICS WITH JUPYTER NOTEBOOKS

EXERCISE 6: APPLYING LINEAR MODELS WITH

SEABORN AND SCIKIT- LEARN

ACTIVITY 1: BUILDING A THIRD-ORDER POLYNOMIAL MODEL

USING CATEGORICAL FEATURES FOR SEGMENTATION ANALYSIS

EXERCISE 7: CREATING CATEGORICAL FIELDS FROM CONTINUOUS VARIABLES AND MAKE SEGMENTED VISUALIZATIONS

SUMMARY

Data Cleaning and Advanced

Machine Learning

INTRODUCTION

PREPARING TO TRAIN A PREDICTIVE MODEL

DETERMINING A PLAN FOR PREDICTIVE ANALYTICS

EXERCISE 8: EXPLORE DATA PREPROCESSING TOOLS AND METHODS

ACTIVITY 2: PREPARING TO TRAIN A PREDICTIVE MODEL FOR THE EMPLOYEE-RETENTION PROBLEM

TRAINING CLASSIFICATION

MODELS

INTRODUCTION TO CLASSIFICATION ALGORITHMS

EXERCISE 9: TRAINING TWO-FEATURE CLASSIFICATION MODELS WITH SCIKIT- LEARN

THE PLOT_DECISION_REGION S FUNCTION

EXERCISE 10: TRAINING K-NEAREST NEIGHBORS FOR OUR MODEL

EXERCISE 11: TRAINING A RANDOM FOREST

ASSESSING MODELS

**WITH K-FOLD CROSS-
VALIDATION AND
VALIDATION CURVES**

**EXERCISE 12: USING K-
FOLD CROSS VALIDATION
AND VALIDATION CURVES
IN PYTHON WITH SCIKIT-
LEARN**

**DIMENSIONALITY
REDUCTION TECHNIQUES**

**EXERCISE 13: TRAINING A
PREDICTIVE MODEL FOR
THE EMPLOYEE
RETENTION PROBLEM**

SUMMARY

**Web Scraping and
Interactive
Visualizations**

INTRODUCTION

**SCRAPING WEB PAGE
DATA**

**INTRODUCTION TO HTTP
REQUESTS**

**MAKING HTTP REQUESTS
IN THE JUPYTER
NOTEBOOK**

**EXERCISE 14: HANDLING
HTTP REQUESTS WITH
PYTHON IN A JUPYTER
NOTEBOOK**

**PARSING HTML IN THE
JUPYTER NOTEBOOK**

**EXERCISE 15: PARSING
HTML WITH PYTHON IN A
JUPYTER NOTEBOOK**

ACTIVITY 3: WEB SCRAPING WITH JUPYTER NOTEBOOKS

INTERACTIVE VISUALIZATIONS

BUILDING A DATAFRAME TO STORE AND ORGANIZE DATA

EXERCISE 16: BUILDING AND MERGING PANDAS DATAFRAMES

INTRODUCTION TO BOKEH

EXERCISE 17: INTRODUCTION TO INTERACTIVE VISUALIZATION WITH BOKEH

ACTIVITY 4: EXPLORING DATA WITH INTERACTIVE VISUALIZATIONS

SUMMARY

Appendix A

Preface

About

This section briefly introduces the author, the coverage of this book, the technical skills you'll need to get started, and the hardware and software requirements required to complete all of the included activities and exercises.

About the Book

Applied Data Science with Python and Jupyter teaches you the skills you need for entry-level data science. You'll learn about some of the most commonly used libraries that are part of the Anaconda distribution, and then explore machine learning models with real datasets to give you the skills and exposure you need for the real world. You'll finish up by learning how easy it can be to scrape and gather your own data from the open web so that you can apply your new skills in an actionable ...

Jupyter Fundamentals

Learning Objectives

By the end of this chapter, you will be able to:

- Describe Jupyter Notebooks and how they are used for data analysis
- Describe the features of Jupyter Notebooks
- Use Python data science libraries
- Perform simple exploratory data analysis

In this chapter, you will learn and implement the fundamental features of the Jupyter notebook by completing several hands-on exercises.

Introduction

Jupyter Notebooks are one of the most important tools for data scientists using Python. This is because they're an ideal environment for developing reproducible data analysis pipelines. Data can be loaded, transformed, and modeled all inside a single Notebook, where it's quick and easy to test out code and explore ideas along the way. Furthermore, all of this can be documented "inline" using formatted text, so you can

make notes for yourself or even produce a structured report.

Other comparable platforms - for example, RStudio or Spyder - present the user with multiple windows, which promote arduous tasks such as copy and pasting code around and rerunning code that has already been executed. These tools also tend to involve **Read Eval Prompt Loops (REPLs)** where code is run in a terminal session that has saved memory. This type of development environment is bad for reproducibility and not ideal for development either. Jupyter Notebooks solve all these issues by giving the user a single window where code snippets are executed and outputs are displayed inline. This lets users develop code efficiently and allows them to look back at previous work for reference, or even to make alterations.

We'll start the chapter by explaining exactly what Jupyter Notebooks are and continue to discuss why they are so popular among data scientists. Then, we'll open a Notebook together and go through some exercises to learn how the platform is used. Finally, we'll dive into our first analysis and perform an exploratory analysis in

Basic Functionality and Features

In this section, we first demonstrate the usefulness of Jupyter Notebooks with examples and through discussion. Then, in order to cover the fundamentals of Jupyter Notebooks for beginners, we'll see the basic usage of them in terms of launching and interacting with the platform. For those who have used Jupyter Notebooks before, this will be mostly a review; however, you will certainly see new things in this topic as well.

WHAT IS A JUPYTER NOTEBOOK AND WHY IS

IT USEFUL?

Jupyter Notebooks are locally run web applications which contain live code, equations, figures, interactive apps, and **Markdown** text. The standard language is Python, and that's what we'll be using for this book; however, note that a variety of alternatives are supported. This includes the other dominant data science language, R:

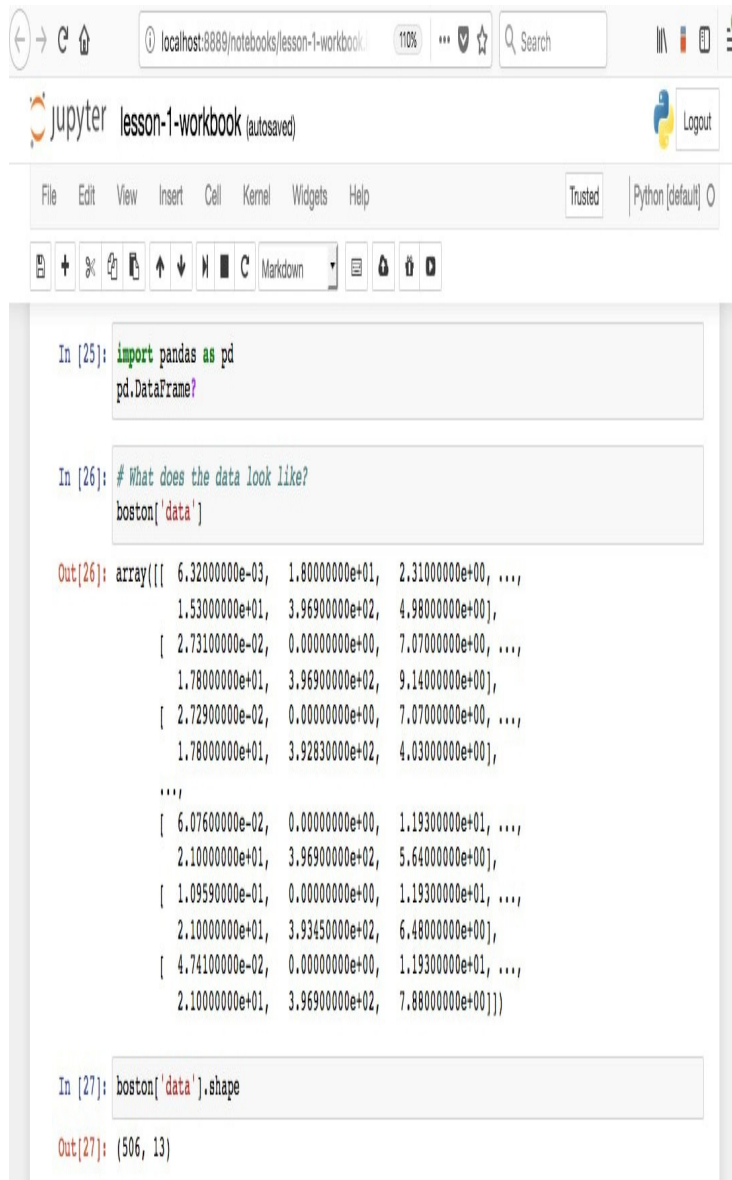


Figure 1.1: Jupyter Notebook sample workbook

Those familiar with R will know about R Markdown.

Markdown documents allow for Markdown-formatted text to be combined with executable code. **Markdown** is a simple language used for styling text on the web. For example, most GitHub repositories have a **README.md** **Markdown** file. This format is useful for basic text formatting. It's comparable to HTML but allows for much less customization.

Commonly used symbols in **Markdown** include hashes (#) to make text into a heading, square and round brackets to insert hyperlinks, and stars to create italicized or bold text:

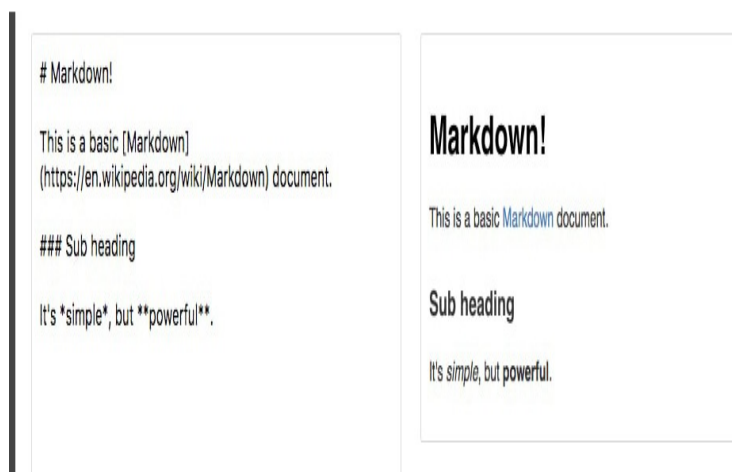


Figure 1.2: Sample Markdown document

Having seen the basics of Markdown, let's come back to R Markdown, where **Markdown** text can be written alongside executable code. Jupyter Notebooks offer the equivalent functionality for Python, although, as we'll see, they function quite differently than R **Markdown** documents. For example, R **Markdown** assumes you are writing **Markdown** unless otherwise specified, whereas Jupyter Notebooks assume you are inputting code. This makes it more appealing to use Jupyter Notebooks for rapid development and testing.

From a data science perspective, there are two primary types for a Jupyter Notebook depending on how they are used: lab-style and deliverable.

Lab-style Notebooks are meant to serve as the programming analog of research journals. These should contain all the work

you've done to load, process, analyze, and model the data. The idea here is to document everything you've done for future reference, so it's usually not advisable to delete or alter previous lab-style Notebooks. It's also a good idea to accumulate multiple date-stamped versions of the Notebook as you progress through the analysis, in case you want to look back at previous states.

Deliverable Notebooks are intended to be presentable and should contain only select parts of the lab-style Notebooks. For example, this could be an interesting discovery to share with your colleagues, an in-depth report of your analysis for a manager, or a summary of the key findings for stakeholders.

In either case, an important concept is reproducibility. If you've been diligent in documenting your software versions, anyone receiving the reports will be able to rerun the Notebook and compute the same results as you did. In the scientific community, where reproducibility is becoming increasingly difficult, this is a breath of fresh air.

NAVIGATING THE PLATFORM

Now, we are going to open up a Jupyter Notebook and start to learn the interface. Here, we will assume you have no prior knowledge of the platform and go over the basic usage.

EXERCISE 1: INTRODUCING JUPYTER NOTEBOOKS

1. Navigate to the companion material directory in the terminal

Note

*Unix machines such as Mac or Linux, command-line navigation can be done using **ls** to display directory contents and **cd** to change directories. On Windows machines, use **dir** to display directory contents and use **cd** to change directories instead. If, for example, you want to change the drive from C: to D:, you should execute **d: to change drives**.*

2. Start a new local Notebook server here by typing the following into the terminal:

```
jupyter notebook
```

A new window or tab of your default browser will open the Notebook Dashboard to the working directory. Here, you will see a list of folders and files contained therein.

3. Click on a folder to navigate to that particular path and open a file by clicking on it. Although its main use is editing IPYNB Notebook files, Jupyter functions as a

standard text editor as well.

4. Reopen the terminal window used to launch the app. We can see the **NotebookApp** being run on a local server. In particular, you should see a line like this:

```
[I 20:03:01.045
NotebookApp] The
Jupyter Notebook is
running at:
http://localhost:8888/
?
token=e915bb06866f19ce
462d959a9193a94c7c088e
81765f9d8a
```

Going to that HTTP address will load the app in your browser window, as was done automatically when starting the app. Closing the window does not stop the app; this should be done from the terminal by typing *Ctrl + C*.

5. Close the app by typing *Ctrl + C* in the terminal. You may also have to confirm by entering **y**. Close the web browser window as well.
6. Load the list of available options by

running the following code:

```
jupyter notebook --  
help
```

7. Open the **NotebookApp** at local port **9000** by running the following:

```
jupyter notebook --  
port 9000
```

8. Click **New** in the upper-right corner of the Jupyter Dashboard and select a kernel from the drop-down menu (that is, select something in the **Notebooks** section):

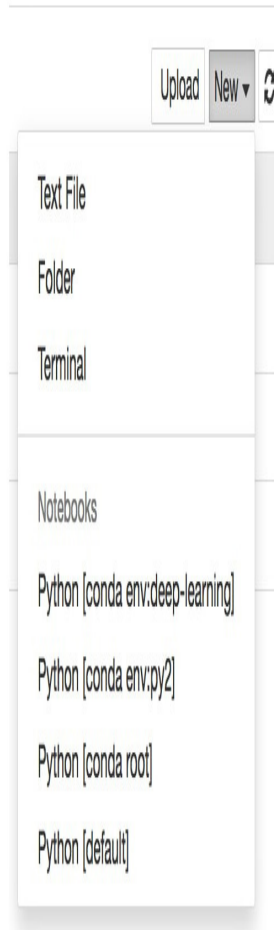


Figure 1.3: Selecting a kernel from the drop down menu

This is the primary method of creating a new Jupyter Notebook.

Kernels provide programming language support for the Notebook. If you have installed Python with Anaconda, that version should be the default kernel. Conda virtual environments will also be available here.

Note

Virtual environments are a great tool for managing multiple projects on the same machine. Each virtual environment may contain a different version of Python and external libraries. Python has built-in virtual environments; however, the Conda virtual environment integrates better with Jupyter Notebooks and boasts other nice features. The documentation is available at:

<https://conda.io/docs/user-guide/tasks/manage-environments.html>.

9. With the newly created blank Notebook, click the top cell and type **print('hello world')**, or any other code snippet that writes to the screen.
10. Click the cell and press *Shift + Enter* or select **Run Cell** in the **Cell** menu.

Any **stdout** or **stderr** output from the code will be displayed beneath as the cell runs.

Furthermore, the string representation of the object written in the final line will be displayed as well. This is very handy, especially for displaying tables, but sometimes we don't want the final object to be displayed. In such cases, a semicolon (;) can be added to the end of the line to suppress the display. New cells expect and run code input by default; however, they can be changed to render **Markdown** instead.

11. Click an empty cell and change it to accept the Markdown-formatted text. This can be done from the drop-down menu icon in the toolbar or by selecting **Markdown** from the **Cell** menu. Write some text in here (any text will do), making sure to utilize Markdown formatting symbols such as #.
12. Scroll to the **Play** icon in the toolbar:



Figure 1.4: Jupyter Notebook tool bar

This can be used to run cells. As we'll see later, however, it's handier to use the keyboard shortcut *Shift + Enter* to run cells.

Right next to this is a **Stop** icon, which can be used to stop cells from running. This is useful, for example, if a cell is taking too long to run:

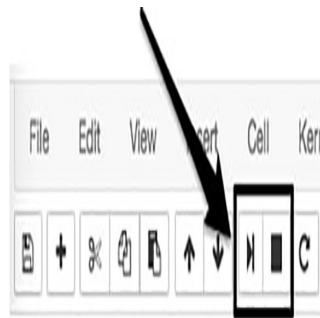


Figure 1.5: Stop icon in Jupyter Notebooks

New cells can be manually added from the **Insert** menu:



Figure 1.6: Adding new cells from the Insert menu in Jupyter Notebooks

Cells can be copied, pasted, and deleted using icons or by selecting options from the **Edit** menu:



Figure 1.7: Edit Menu in the Jupyter Notebooks



Figure 1.8: Cutting and copying cells in Jupyter Notebooks

Cells can also be moved up and down this way:

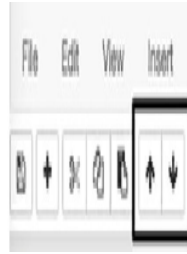


Figure 1.9: Moving cells up and down in Jupyter Notebooks

There are useful options under the **Cell** menu to run a group of cells or the entire Notebook:



Figure 1.10: Running cells in Jupyter Notebooks

Experiment with the toolbar options to move cells up and down, insert new cells, and delete cells. An important thing to understand about these Notebooks is the shared memory between cells. It's quite simple: every cell existing on the sheet has access to the global set of variables. So, for example, a function defined in one cell could be called from any other, and the same applies to variables. As one would expect, anything within the scope of a function will not be a global variable and can only be accessed from within that specific function.

13. Open the **Kernel** menu to see the selections. The **Kernel** menu is useful for stopping script executions and restarting the Notebook if the kernel dies. Kernels can also be swapped here at any time, but it is inadvisable to use multiple kernels for a single Notebook due to reproducibility concerns.
14. Open the **File** menu to see the selections. The **File** menu contains

options for downloading the Notebook in various formats. In particular, it's recommended to save an HTML version of your Notebook, where the content is rendered statically and can be opened and viewed "as you would expect" in web browsers.

The Notebook name will be displayed in the upper-left corner. New Notebooks will automatically be named **Untitled**.

15. Change the name of your IPYNB Notebook file by clicking on the current name in the upper-left corner and typing the new name. Then, save the file.
16. Close the current tab in your web browser (exiting the Notebook) and go to the **Jupyter Dashboard** tab, which should still be open. (If it's not open, then reload it by copy and pasting the HTTP link from the terminal.)

Since we didn't shut down the Notebook, and we just saved and exited, it will have a green book

symbol next to its name in the **Files** section of the Jupyter Dashboard and will be listed as **Running** on the right side next to the last modified date. Notebooks can be shut down from here.

17. Quit the Notebook you have been working on by selecting it (checkbox to the left of the name), and then click the orange **Shutdown** button:

Note

Read through the basic keyboard shortcuts and test them.

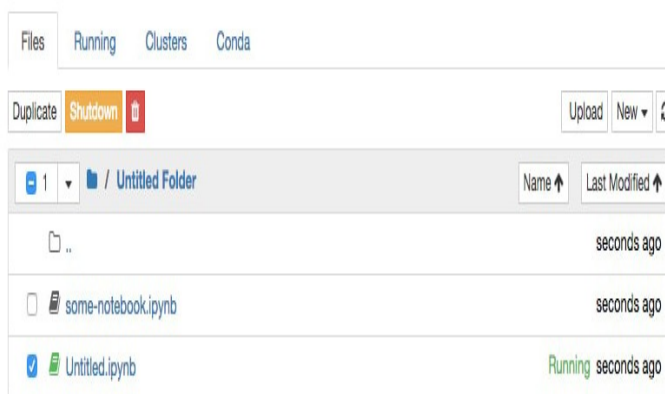


Figure 1.11: Shutting down the Jupyter notebook

Note

If you plan to spend a lot of time working with Jupyter Notebooks, it's worthwhile to learn the keyboard shortcuts.

*This will speed up your workflow considerably. Particularly useful commands to learn are the shortcuts for manually adding new cells and converting cells from code to Markdown formatting. Click on **Keyboard Shortcuts** from the **Help** menu to see how.*

JUPYTER FEATURES

Jupyter has many appealing features that make for efficient Python programming. These include an assortment of things, from methods for viewing docstrings to executing Bash commands. We will explore some of these features in this section.

Note

The official IPython documentation can be found here: <http://ipython.readthedocs.io/en/stable/>. It has details on the features we will discuss here and others.

EXERCISE 2: IMPLEMENTING JUPYTER'S MOST USEFUL FEATURES

1. Navigate to the **lesson-1** directory from the Jupyter Dashboard and open **lesson-1-workbook.ipynb** by selecting it.

The standard file extension for Jupyter Notebooks is **.ipynb**,

which was introduced back when they were called IPython Notebooks.

2. Scroll down to **Subtopic C: Jupyter Features** in the Jupyter Notebook.

We start by reviewing the basic keyboard shortcuts. These are especially helpful to avoid having to use the mouse so often, which will greatly speed up the workflow.

You can get help by adding a question mark to the end of any object and running the cell. Jupyter finds the docstring for that object and returns it in a pop-out window at the bottom of the app.

3. Run the **Getting Help** cell and check how Jupyter displays the docstrings at the bottom of the Notebook. Add a cell in this section and get help on the object of your choice:

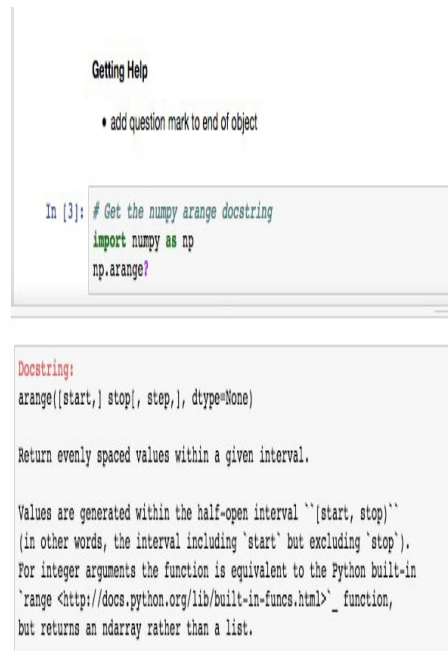


Figure 1.12: Getting help in Jupyter Notebooks

4. Click an empty code cell in the **Tab Completion** section. Type import (including the space after) and then press the **Tab** key:

Tab Completion

Example of Jupyter tab completion include:

- listing available modules on import

```
import <tab>
```

```
from numpy import <tab>
```

- listing available modules after import

```
np.<tab>
```

Figure 1.13: Tab completion in Jupyter

Notebooks

The above action listed all the available modules for import.

Tab completion can be used for the following: **list available modules when importing external libraries; list available modules of imported external libraries; function and variable completion.**

This can be especially useful when you need to know the available input arguments for a module, when exploring a new library, to discover new modules, or simply to speed up workflow. They will save time writing out variable names or functions and reduce bugs from typos. The tab completion works so well that you may have difficulty coding Python in other editors after today!

5. Scroll to the Jupyter Magic

Functions section and run the cells containing **%lsmagic** and **%matplotlib** inline:

Jupyter Magic Functions

List of the available magic commands:

```
%lsmagic
```

Available line magics:

```
%alias %alias_magic %autocall %automagic %auto  
ist %dirs %doctest_mode %ed %edit %env %gui  
dpy %logoff %logon %logstart %logstate %logst  
ook %page %pastebin %pdb %pdef %pdoc %pfile  
ushd %pwd %pycat %pylab %qtconsole %quickref  
%run %save %sc %set_env %store %sx %system
```

Figure 1.14: Jupyter Magic functions

The percent signs, % and %, are one of the basic features of Jupyter Notebook and are called magic commands. Magics starting with %% will apply to the entire cell, and magics starting with % will only apply to that line.

%lsmagic lists the available options. We will discuss and show examples of some of the most useful ones. The most common magic command you will probably see is **%matplotlib inline**, which allows matplotlib figures to be displayed in the Notebook without having to explicitly use **plt.show()**.

The timing functions are very handy and come in two varieties: a standard timer (**%time** or

`%%time`) and a timer that measures the average runtime of many iterations (`%timeit` and `%%timeit`).

Note

Notice how list comprehensions are quicker than loops in Python. This can be seen by comparing the wall time for the first and second cell, where the same calculation is done significantly faster with the list comprehension.

6. Run the cells in the **Timers** section.

Note the difference between using one and two percent signs. Even by using a Python kernel (as you are currently doing), other languages can be invoked using magic commands. The built-in options include JavaScript, R, Pearl, Ruby, and Bash. Bash is particularly useful, as you can use Unix commands to find out where you are currently (**pwd**), what's in the directory (**ls**), make new folders (**mkdir**), and write file contents (**cat/head/tail**).

7. Run the first cell in the **Using bash** in the notebook section.

This cell writes some text to a file in the working directory, prints the directory contents, prints an empty line, and then writes back the contents of the newly created file before removing it:

```
Using bash in the notebook

In [9]: %%bash

echo "using bash from inside Jupyter!" > test-file.txt
ls
echo ""
cat test-file.txt
rm test-file.txt

Lesson 1
Lesson 1.docx
Lesson 1.pptx
lesson-1-workbook.html
lesson-1-workbook.ipynb
test-file.txt
~$sson 1.docx

using bash from inside Jupyter!
```

Figure 1.15: Using Bash in Jupyter Notebooks

8. Run the cells containing only **ls** and **pwd**.

Note how we did not have to explicitly use the Bash magic command for these to work. There are plenty of external magic commands that can be installed. A popular one is **ipython-sql**,

which allows for SQL code to be executed in cells.

9. Open a new terminal window and execute the following code to install `ipython-sql`:

```
pip install ipython-sql
```

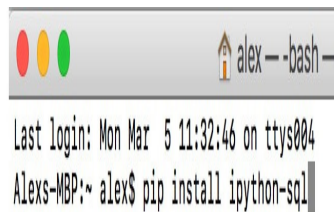


Figure 1.16: Installing `ipython-sql` using `pip`

10. Run the `%load_ext sql` cell to load the external command into the Notebook:

```
: # Source: https://github.com/catherinedevlin/ipython-sql
# do pip install ipython-sql in the terminal
%load_ext sql
```

Figure 1.17: Loading `sql` in Jupyter Notebooks

This allows for connections to remote databases so that queries can be executed (and thereby documented) right inside the

Notebook.

11. Run the cell containing the SQL sample query:

```
%%sql sqlite://

SELECT *
FROM (
    SELECT 'Hello' as msg_1
) A JOIN (
    SELECT 'World!' as msg_2
) B;
```

Done.

msg_1	msg_2
Hello	World!

Figure 1.18: Running a sample SQL query

Here, we first connect to the local sqlite source; however, this line could instead point to a specific database on a local or remote server. Then, we execute a simple **SELECT** to show how the cell has been converted to run SQL code instead of Python.

12. Install the version documentation tool now from the terminal using **pip**. Open up a new window and run the following code:

```
pip install
version_information
```

Once installed, it can then be imported into any Notebook using **%load_ext version_information**.

Finally, once loaded, it can be used to display the versions of each piece of software in the Notebook.

The **%version_information commands** helps with **documentation**, but it does not come as standard with Jupyter. Like the SQL example we just saw, it can be installed from the command line with **pip**.

13. Run the cell that loads and calls the **version_information** command:

```
%load_ext version_information
%version_information requests, numpy, pandas, matplotlib, seaborn, sklearn
```

Software	Version
Python	3.5.4 64bit [GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)]
IPython	6.1.0
OS	Darwin 16.5.0 x86_64 1386 64bit
requests	2.18.4
numpy	1.13.1
pandas	0.20.3
matplotlib	2.0.2
seaborn	0.8.0
sklearn	0.19.0
Wed Oct 11 19:46:08 2017 PDT	

Figure 1.19: Version Information in Jupyter

CONVERTING A JUPYTER NOTEBOOK TO A PYTHON SCRIPT

You can convert a Jupyter Notebook to a Python script. This is equivalent to copying and pasting the contents of each code cell into a single **.py** file. The Markdown sections are also included as comments.

The conversion can be done from the **NotebookApp** or in the command line as follows:

```
jupyter nbconvert --to=python lesson-1-notebook.ipynb
```

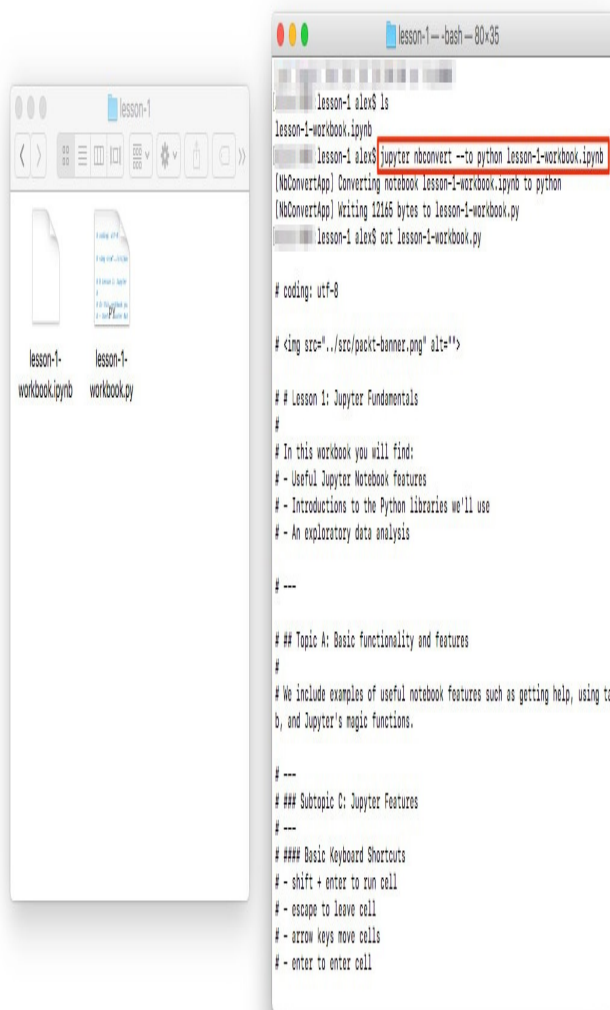


Figure 1.20: Converting a Jupyter Notebook into a Python Script

This is useful, for example, when you want to determine the library requirements for a Notebook using a tool such as **pipreqs**. This tool determines the libraries used in a project and exports them into a **requirements.txt** file (and it can be installed by running `pip install pipreqs`).

The command is called from outside the folder containing your **.py** files. For example, if the **.py** files are inside a folder called **lesson-1**, you could do the following:

```
pipreqs lesson-1/
```

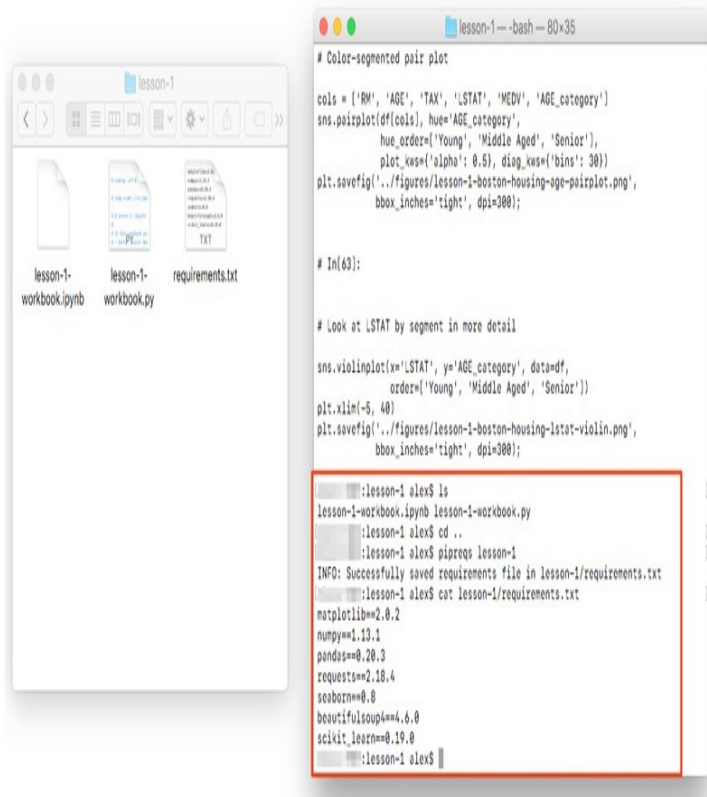


Figure 1.21: Determining library requirements using pipreqs

The resulting **requirements.txt** file for **lesson-1-workbook.ipynb** looks like this:

```

cat lesson-1/requirements.txt

matplotlib==2.0.2 numpy==1.13.1

pandas==0.20.3

requests==2.18.4

seaborn==0.8

beautifulsoup4==4.6.0

scikit_learn==0.19.0

```

PYTHON LIBRARIES

Having now seen all the basics of Jupyter Notebooks, and even

some more advanced features, we'll shift our attention to the Python libraries we'll be using in this book. Libraries, in general, extend the default set of Python functions. Examples of commonly used standard libraries are **datetime**, **time**, and **os**. These are called standard libraries because they come standard with every installation of Python.

For data science with Python, the most important libraries are external, which means they do not come standard with Python.

The external data science libraries we'll be using in this book are NumPy, Pandas, Seaborn, matplotlib, scikit-learn, Requests, and Bokeh.

Note

A word of caution: It's a good idea to import libraries using industry standards, for example, `import numpy as np`; this way, your code is more readable. Try to avoid doing things such as `from numpy import *`, as you may unwittingly overwrite functions. Furthermore, it's often nice to have modules linked to the library via a dot (.) for code readability.

Let's briefly introduce each.

- **NumPy** offers multi-dimensional data structures (arrays) on which operations can be performed far quicker than standard Python data structures (for example, lists). This is done in part by performing operations in the background using C. NumPy also offers various mathematical and data manipulation functions.
- **Pandas** is Python's answer to the R

DataFrame. It stores data in 2D tabular structures where columns represent different variables and rows correspond to samples. Pandas provides many handy tools for data wrangling such as filling in **NaN** entries and computing statistical descriptions of the data. Working with Pandas DataFrames will be a big focus of this book.

- **Matplotlib** is a plotting tool inspired by the MATLAB platform. Those familiar with R can think of it as Python's version of ggplot. It's the most popular Python library for plotting figures and allows for a high level of customization.
- **Seaborn** works as an extension to matplotlib, where various plotting tools useful for data science are included. Generally speaking, this allows for analysis to be done much faster than if you were to create the same things *manually* with libraries such as matplotlib and scikit-learn.
- **scikit-learn** is the most commonly used machine learning library. It offers top-of-the-line algorithms

and a very elegant API where models are instantiated and then *fit* with data. It also provides data processing modules and other tools useful for predictive analytics.

- **Requests** is the go-to library for making HTTP requests. It makes it straightforward to get HTML from web pages and interface with APIs. For parsing the HTML, many choose BeautifulSoup4, which we will also cover in this book.
- **Bokeh** is an interactive visualization library. It functions similar to matplotlib, but allows us to add hover, zoom, click, and use other interactive tools to our plots. It also allows us to render and play with the plots inside our Jupyter Notebook.

Having introduced these libraries, let's go back to our Notebook and load them, by running the **import** statements. This will lead us into our first analysis, where we finally start working with a dataset.

EXERCISE 3: IMPORTING THE EXTERNAL

LIBRARIES AND SETTING UP THE PLOTTING ENVIRONMENT

1. Open up the **lesson 1** Jupyter Notebook and scroll to the **Subtopic D: Python Libraries** section.

Just like for regular Python scripts, libraries can be imported into the Notebook at any time. It's best practice to put the majority of the packages you use at the top of the file. Sometimes it makes sense to load things midway through the Notebook and that is completely fine.

2. Run the cells to import the external libraries and set the plotting options:

```
# Common standard libraries
```

```
import datetime
```

```
import time
```

```
import os
```

```
# Common external libraries
```

```
import pandas as pd
```

```
import numpy as np
```

```
import sklearn # scikit-learn
```

```
import requests
```

```
from bs4 import BeautifulSoup
```

Figure 1.22: Importing Python libraries

For a nice Notebook setup, it's often useful to set various options along with the imports at the top. For example, the following can be run to change the figure appearance to something more aesthetically pleasing than the matplotlib and Seaborn defaults:

```
import matplotlib.pyplot as plt
```

```
%matplotlib inline import
```

```
seaborn as sns
```

```
# See here for more options:
```

```
https://matplotlib.org/users/  
customizing.html
```

```
%config
```

```
InlineBackend.figure_format='retina'
```

```
sns.set() # Revert to matplotlib
```

```
defaults

plt.rcParams['figure.figsize'] = (9, 6)

plt.rcParams['axes.labelpad'] = 10
sns.set_style("darkgrid")
```

So far in this book, we've gone over the basics of using Jupyter Notebooks for data science. We started by exploring the platform and finding our way around the interface. Then, we discussed the most useful features, which include tab completion and magic functions. Finally, we introduced the Python libraries we'll be using in this book.

The next section will be very interactive as we perform our first analysis together using the Jupyter Notebook.

Our First Analysis - The Boston Housing Dataset

So far, this chapter has focused on the features and basic usage of Jupyter. Now, we'll put this into practice and do some data exploration and analysis.

The dataset we'll look at in this section is the so-called Boston housing dataset. It contains US census data concerning houses in various areas around the city of Boston. Each sample corresponds to a unique area and has about a dozen measures. We should think of samples as rows and measures as columns. The data was first published in 1978 and is quite small, containing only about 500 samples.

Now that we know something about the context of the dataset, let's decide on a rough plan for the exploration and analysis. If applicable, this plan would accommodate the relevant question(s) under study. In this case, the goal is not to answer a question but to instead show Jupyter in action and illustrate

some basic data analysis methods.

Our general approach to this analysis will be to do the following:

- Load the data into Jupyter using a Pandas DataFrame
- Quantitatively understand the features
- Look for patterns and generate questions
- Answer the questions to the problems

LOADING THE DATA INTO JUPYTER USING A PANDAS DATAFRAME

Oftentimes, data is stored in tables, which means it can be saved as a **comma-separated variable (CSV)** file. This format, and many others, can be read into Python as a DataFrame object, using the Pandas library. Other common formats include **tab-separated variable (TSV)**, SQL tables, and JSON data structures. Indeed, Pandas has support for all of these. In this example, however, we are not going to load the data this way because the dataset is available directly through scikit-learn.

Note

An important part after loading data for analysis is ensuring that it's clean. For example, we would generally need to deal

with missing data and ensure that all columns have the correct datatypes. The dataset we use in this section has already been cleaned, so we will not need to worry about this. However, we'll see messier data in the second chapter and explore techniques for dealing with it.

EXERCISE 4: LOADING THE BOSTON HOUSING DATASET

1. Scroll to **Subtopic A** of **Topic B: Our first Analysis: the Boston Housing Dataset** in chapter 1 of the Jupyter Notebook.

The Boston housing dataset can be accessed from the **sklearn.datasets** module using the **load_boston** method.

2. Run the first two cells in this section to load the Boston dataset and see the **datastructures** type:

```
from sklearn import datasets  
boston = datasets.load_boston()
```

```
type(boston)
```

```
sklearn.utils.Bunch
```

Figure 1.23: Loading the Boston dataset

The output of the second cell tells us that it's a scikit-learn **Bunch** object. Let's get some more information about that to understand what we are dealing with.

3. Run the next cell to import the base object from scikit-learn **utils** and print the docstring in our Notebook:

```
In [4]: from sklearn.utils import Bunch
        Bunch?

Init signature: Bunch(**kwargs)
Docstring:
Container object for datasets

Dictionary-like object that exposes its keys as attributes.

>>> b = Bunch(a=1, b=2)
>>> b['b']
2
```

Figure 1.24: Importing base objects and printing the docstring

4. Print the field names (that is, the keys to the dictionary) by running the next cell. We find these fields to be self-explanatory: **['DESCR', 'target', 'data', 'feature_names']**.
5. Run the next cell to print the dataset description contained in **boston['DESCR']**.

Note that in this call, we explicitly want to print the field value so that the Notebook renders the content in

a more readable format than the string representation (that is, if we just type `boston['DESCR']` without wrapping it in a `print` statement). We then see the dataset information as we've previously summarized:

```
Boston House Prices
dataset
```

```
=====
=====
```

```
Notes
```

```
-----
```

```
Data Set
```

```
Characteristics:
```

```
:Number of Instances:
506
```

```
:Number of Attributes:
13 numeric/categorical
predictive
```

```
:Median Value
(attribute 14) is
usually the target
```

```
:Attribute Information
(in order):
```

```
- CRIM per capita
```

crime rate by town

...

...

- MEDV Median value of
owner-occupied homes
in \$1000's

:Missing Attribute
Values: None

Note

*Briefly read through the feature descriptions and/or describe them yourself. For the purposes of this tutorial, the most important fields to understand are **RM**, **AGE**, **LSTAT**, and **MEDV**. Note down the important variables that we will use in the dataset, such as **RM**, **AGE**, **LSTAT**, and **MEDV**.*

Of particular importance here are the feature descriptions (under **Attribute Information**). We will use this as reference during our analysis.

Note

For the complete code, refer to the

following: <https://bit.ly/2EL11cW>

Now, we are going to create a Pandas DataFrame that contains the data. This is beneficial for a few reasons: all of our data will be contained in one object, there are useful and computationally efficient DataFrame methods we can use, and other libraries such as Seaborn have tools that integrate nicely with DataFrames.

In this case, we will create our DataFrame with the standard constructor method.

6. Run the cell where Pandas is imported and the docstring is retrieved for **pd.DataFrame**:

```
In [5]: import pandas as pd
        pd.DataFrame?
```

Init signature: pd.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)

Docstring:

Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects. The primary pandas data structure

Parameters

data : numpy ndarray (structured or homogeneous), dict, or DataFrame
Dict can contain Series, arrays, constants, or list-like objects

index : Index or array-like
Index to use for resulting frame. Will default to np.arange(n) if no indexing information part of input data and no index provided

columns : Index or array-like
Column labels to use for resulting frame. Will default to np.arange(n) if no column labels are provided

**Figure 1.25: Retrieving the docstring for
pd.DataFrame**

The docstring reveals the DataFrame input parameters. We want to feed in **boston['data']** for the data and use **boston['feature_names']** for the headers.

7. Run the next few cells to print the data, its shape, and the feature names:

```
# What does the data look like?
```

```
boston['data']
```

```
array([[ 6.32000000e-03,  1.80000000e+01,  2.31000000e+00, ...,
         1.53000000e+01,  3.96900000e+02,  4.98000000e+00],
       [ 2.73100000e-02,  0.00000000e+00,  7.07000000e+00, ...,
         1.78000000e+01,  3.96900000e+02,  9.14000000e+00],
       [ 2.72900000e-02,  0.00000000e+00,  7.07000000e+00, ...,
         1.78000000e+01,  3.92830000e+02,  4.03000000e+00],
       ...,
       [ 6.07600000e-02,  0.00000000e+00,  1.19300000e+01, ...,
         2.10000000e+01,  3.96900000e+02,  5.64000000e+00],
       [ 1.09590000e-01,  0.00000000e+00,  1.19300000e+01, ...,
         2.10000000e+01,  3.93450000e+02,  6.48000000e+00],
       [ 4.74100000e-02,  0.00000000e+00,  1.19300000e+01, ...,
         2.10000000e+01,  3.96900000e+02,  7.88000000e+00]])
```

```
boston['data'].shape
```

```
(506, 13)
```

```
boston['feature_names']
```

```
array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT'],
      dtype='<U7')
```

Figure 1.26: Printing data, shape, and feature names

Looking at the output, we see that our data is in a 2D NumPy array.

Running the command

`boston['data'].shape`

returns the length (number of samples) and the number of features as the first and second outputs, respectively.

8. Load the data into a Pandas DataFrame **df** by running the following:

```
df =  
pd.DataFrame(data=boston['data'],  
              columns=boston['feature_names'])
```

In machine learning, the variable that is being modeled is called the target variable; it's what you are trying to predict given the features. For this dataset, the suggested target is **MEDV**, the median house value in 1,000s of dollars.

9. Run the next cell to see the shape of the target:

```
# Still need to add the target variable
boston['target'].shape

(506,)
```

Figure 1.27: Code for viewing the shape of the target

We see that it has the same length as the features, which is what we expect. It can therefore be added as a new column to the DataFrame.

10. Add the target variable to `df` by running the cell with the following:

```
df['MEDV'] =
boston['target']
```

11. Move the target variable to the front of **df** by running the cell with the following code:

```
y = df['MEDV'].copy()
del df['MEDV']

df = pd.concat((y,
df), axis=1)
```

This is done to distinguish the target from our features by storing it to the front of our DataFrame.

Here, we introduce a dummy variable **y** to hold a copy of the target column before removing it from the DataFrame. We then use the Pandas concatenation function to combine it with the remaining DataFrame along the 1st axis (as opposed to the 0th axis, which combines rows).

Note

*You will often see dot notation used to reference DataFrame columns. For example, previously we could have done **y = df.MEDV.copy()**. This does not work for deleting columns, however; **del df.MEDV** would raise an error.*

12. Implement **df.head()** or **df.tail()** to glimpse the data and **len(df)** to verify that number of samples is what we expect. Run the next few cells to see the head, tail, and length of **df**:

```
df.head()
```

	MEDV	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
0	24.0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90
1	21.6	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90
2	34.7	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83
3	33.4	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63
4	36.2	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90

Figure 1.28: Printing the head of the data frame df

```
df.tail()
```

	MEDV	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
501	22.4	0.06263	0.0	11.93	0.0	0.573	6.583	69.1	2.4706	1.0	273.0	21.0	391.99
502	20.6	0.04527	0.0	11.93	0.0	0.573	6.120	76.7	2.2875	1.0	273.0	21.0	386.90
503	23.9	0.06076	0.0	11.93	0.0	0.573	6.976	91.0	2.1675	1.0	273.0	21.0	386.90
504	22.0	0.10559	0.0	11.93	0.0	0.573	6.794	89.3	2.3889	1.0	273.0	21.0	383.45
505	11.9	0.04741	0.0	11.93	0.0	0.573	6.030	80.8	2.5050	1.0	273.0	21.0	386.90

```
len(df)
```

506

Figure 1.29: Printing the tail of data frame df

Each row is labeled with an index value, as seen in bold on the left side of the table. By default, these are a set of integers starting at 0 and incrementing by one for each row.

13. Printing **df.dtypes** will show the datatype contained within each column. Run the next cell to see the datatypes of each column. For this dataset, we see that every field is a float and therefore most likely a continuous variable, including the target. This means that predicting the target variable is a regression problem.

14. Run **df.isnull()** to clean the dataset as Pandas automatically sets missing data as **NaN** values. To get the number of **NaN** values per column, we can do **df.isnull().sum():**

```
# Identify and NAs  
df.isnull().sum()
```

```
MEDV      0  
CRIM      0  
ZN        0  
INDUS     0  
CHAS      0  
NOX       0  
RM        0  
AGE       0  
DIS       0  
RAD       0  
TAX       0  
PTRATIO   0  
B         0  
LSTAT     0  
dtype: int64
```

**Figure 1.30: Cleaning the dataset by
identifying NaN values**

`df.isnull()` returns a Boolean

frame of the same length as **df**.

For this dataset, we see there are no **NaN** values, which means we have no immediate work to do in cleaning the data and can move on.

15. Remove some columns by running the cell that contains the following code:

```
for col in ['ZN',  
            'NOX', 'RAD',  
            'PTRATIO', 'B']:  
  
    del df[col]
```

This is done to simplify the analysis. We will focus on the remaining columns in more detail.

DATA EXPLORATION

Since this is an entirely new dataset that we've never seen before, the first goal here is to understand the data. We've already seen the textual description of the data, which is important for qualitative understanding. We'll now compute a quantitative description.

EXERCISE 5: ANALYZING THE BOSTON HOUSING DATASET

1. Navigate to **Subtopic B:**
Data exploration in the
Jupyter Notebook and run the cell
containing **`df.describe()`**:

```
df.describe().T
```

	count	mean	std	min	25%	50%	75%	max
MEDV	506.0	22.532806	9.197104	5.00000	17.025000	21.20000	25.000000	50.0000
CRIM	506.0	3.593761	8.596783	0.00632	0.082045	0.25651	3.647423	88.9762
INDUS	506.0	11.136779	6.860353	0.46000	5.190000	9.69000	18.100000	27.7400
CHAS	506.0	0.069170	0.253994	0.00000	0.000000	0.00000	0.000000	1.0000
RM	506.0	6.284634	0.702617	3.56100	5.885500	6.20850	6.823500	8.7800
AGE	506.0	68.574901	28.149861	2.90000	45.025000	77.50000	94.075000	100.0000
DIS	506.0	3.795043	2.105710	1.12960	2.100175	3.20745	5.188425	12.1265
TAX	506.0	408.237154	168.537116	187.00000	279.000000	330.00000	666.000000	711.0000
LSTAT	506.0	12.653063	7.141062	1.73000	6.950000	11.36000	16.955000	37.9700

Figure 1.31: Computation and output of statistical properties

This computes various properties including the mean, standard deviation, minimum, and maximum for each column. This table gives a high-level idea of how everything is distributed. Note that we have taken the transform of the result by adding a `.T` to the output; this swaps the rows and columns.

Going forward with the analysis, we will specify a set of columns to focus on.

2. Run the cell where these "focus columns" are defined:

```
cols = ['RM', 'AGE',  
        'TAX', 'LSTAT',  
        'MEDV']
```

3. Display the aforementioned subset of columns of the DataFrame by running `df[cols].head()`:

```
df[cols].head()
```

	RM	AGE	TAX	LSTAT	MEDV
0	6.575	65.2	296.0	4.98	24.0
1	6.421	78.9	242.0	9.14	21.6
2	7.185	61.1	242.0	4.03	34.7
3	6.998	45.8	222.0	2.94	33.4
4	7.147	54.2	222.0	5.33	36.2

Figure 1.32: Displaying focus columns

As a reminder, let's recall what each of these columns is. From the dataset documentation, we have the following:

- RM average number of rooms per dwelling
- AGE proportion of owner-occupied units built prior to 1940

- TAX full-value property-tax rate per \$10,000
- LSTAT % lower status of the population
- MEDV Median value of owner-occupied homes in \$1000's

To look for patterns in this data, we can start by calculating the pairwise correlations using

pd.DataFrame.corr.

4. Calculate the pairwise correlations for our selected columns by running the cell containing the following code:

```
df[cols].corr()
```

	RM	AGE	TAX	LSTAT	MEDV
RM	1.000000	-0.240265	-0.292048	-0.613808	0.695360
AGE	-0.240265	1.000000	0.506456	0.602339	-0.376955
TAX	-0.292048	0.506456	1.000000	0.543993	-0.468536
LSTAT	-0.613808	0.602339	0.543993	1.000000	-0.737663
MEDV	0.695360	-0.376955	-0.468536	-0.737663	1.000000

Figure 1.33: Pairwise calculation of correlation

This resulting table shows the correlation score between each set of values. Large positive scores indicate a strong positive (that is, in the same direction) correlation. As expected, we see maximum values of 1 on the diagonal.

By default, Pandas calculates the standard correlation coefficient for each pair, which is also called the Pearson coefficient. This is defined as the covariance between two variables, divided by the product of their standard deviations:

$$\rho_{X,Y} = \frac{\text{cov}(X,Y)}{\sigma_X \sigma_Y}$$

The covariance, in turn, is defined as follows:

$$Y = a_0 + a_1 b + a_2 c + a_3 d$$

Here, n is the number of samples, x_i and y_i are the individual samples being summed over, and X and Y

are the means of each set.

Instead of straining our eyes to look at the preceding table, it's nicer to visualize it with a heatmap. This can be done easily with Seaborn.

5. Run the next cell to initialize the plotting environment, as discussed earlier in the chapter. Then, to create the heatmap, run the cell containing the following code:

```
import
matplotlib.pyplot as
plt import seaborn as
sns

%matplotlib inline

ax =
sns.heatmap(df[cols].c
orr(),

cmap=sns.cubehelix_pal
ette(20, light=0.95,
dark=0.15))

ax.xaxis.tick_top() #
move labels to the top

plt.savefig('../figure
s/lesson-1-boston-
housing-corr.png',
```

```
bbox_inches='tight',  
dpi=300)
```

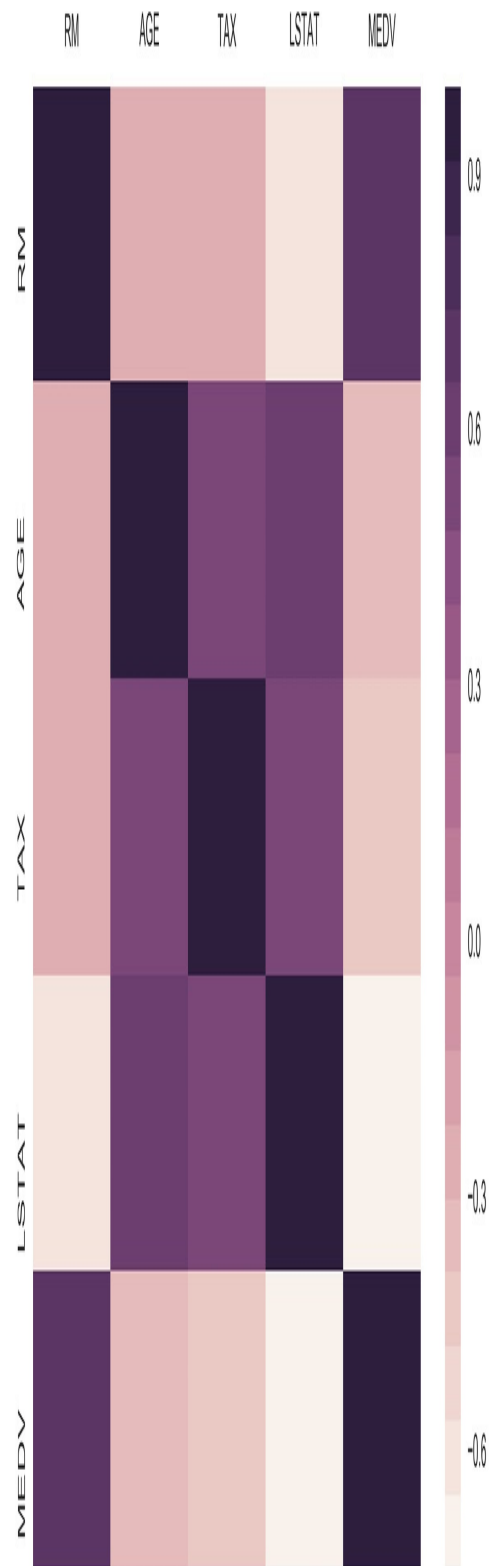


Figure 1.34: Plot of the heat map for all variables

We call **sns.heatmap** and pass the pairwise correlation matrix as input. We use a custom color palette here to override the Seaborn default. The function returns a **matplotlib.axes** object which is referenced by the variable **ax**.

The final figure is then saved as a high resolution PNG to the **figures** folder.

For the final step in our dataset exploration exercise, we'll visualize our data using Seaborn's **pairplot** function.

Visualize the DataFrame using Seaborn's **pairplot** function. Run the cell containing the following code:

```
sns.pairplot(df[cols],  
             plot_kws={'alpha': 0.6},  
             diag_kws={'bins': 30})
```

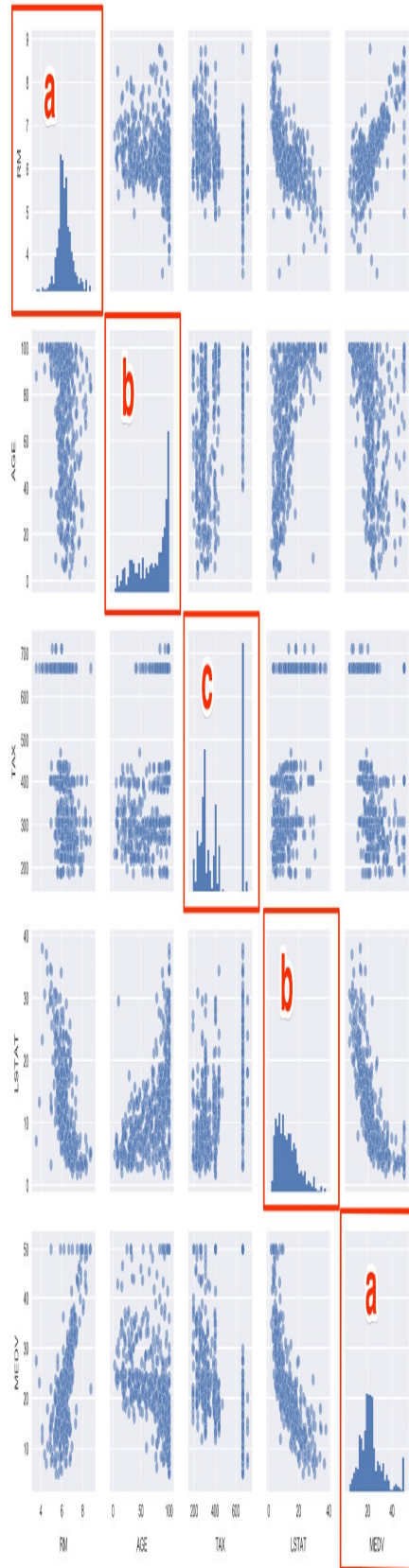


Figure 1.35: Data visualization using Seaborn

Note

Note that unsupervised learning techniques are outside the scope of this book.

Looking at the histograms on the diagonal, we see the following:

a: **RM** and **MEDV** have the closest shape to normal distributions.

b: **AGE** is skewed to the left and **LSTAT** is skewed to the right (this may seem counterintuitive but skew is defined in terms of where the mean is positioned in relation to the max).

c: For **TAX**, we find a large amount of the distribution is around 700. This is also evident from the scatter plots.

Taking a closer look at the **MEDV** histogram in the bottom right, we actually see something similar to **TAX** where there is a large upper-limit bin around \$50,000. Recall when we did **df.describe()**, the min and max of **MEDV** was 5k and 50k, respectively. This suggests that median house values in the dataset were capped at 50k.

INTRODUCTION TO PREDICTIVE ANALYTICS WITH JUPYTER NOTEBOOKS

Continuing our analysis of the Boston housing dataset, we can see that it presents us with a regression problem where we predict a continuous target variable given a set of features. In particular, we'll be predicting the median house value (**MEDV**).

We'll train models that take only one feature as input to make this prediction. This way, the models will be conceptually simple to understand and we can focus more on the technical details of the scikit-learn API. Then, in the next chapter, you'll be more comfortable dealing with the relatively complicated models.

EXERCISE 6: APPLYING LINEAR MODELS WITH SEABORN AND SCIKIT- LEARN

1. Scroll to **Subtopic C:**
**Introduction to
predictive analytics** in
the Jupyter Notebook and look just
above at the pairplot we created in
the previous section. In particular,
look at the scatter plots in the
bottom-left corner:



Figure 1.36: Scatter plots for MEDV and LSTAT

Note how the number of rooms per house (**RM**) and the % of the population that is lower class (**LSTAT**) are highly correlated with the median house value (**MDEV**). Let's pose the following question: how well can we predict **MDEV** given these variables?

To help answer this, let's first visualize the relationships using Seaborn. We will draw the scatter plots along with the line of best fit linear models.

2. Draw scatter plots along with the linear models by running the cell that contains the following:

```
fig, ax =  
plt.subplots(1, 2)  
sns.regplot('RM',  
            'MEDV', df, ax=ax[0],  
            scatter_kws={'alpha':  
0.4}))  
sns.regplot('LSTAT',  
            'MEDV', df, ax=ax[1],  
            scatter_kws={'alpha':
```

0.4}))

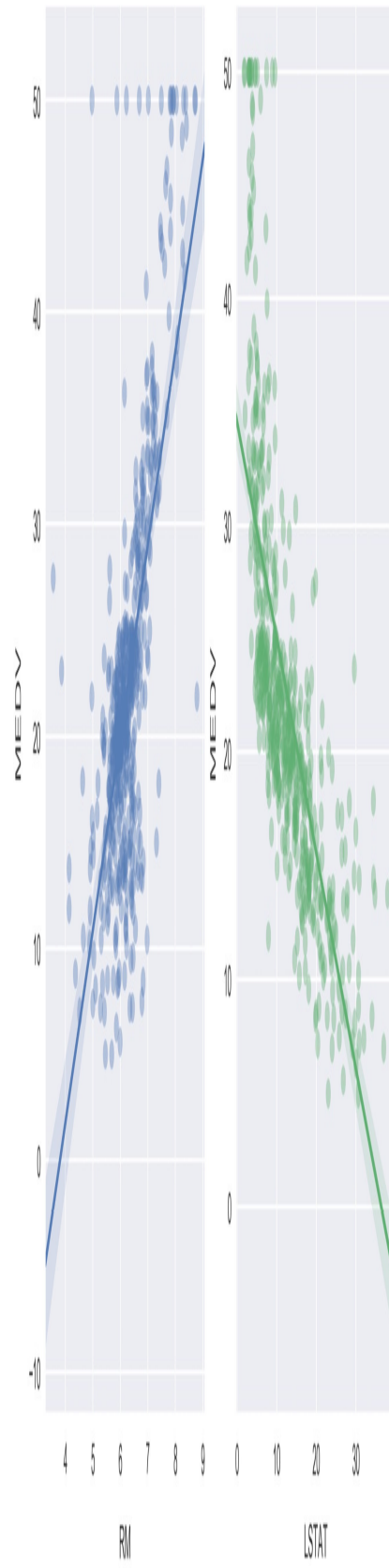


Figure 1.37: Drawing scatter plots using linear models

The line of best fit is calculated by minimizing the ordinary least squares error function, something Seaborn does automatically when we call the **regplot** function. Also note the shaded areas around the lines, which represent 95% confidence intervals.

Note

*These 95% confidence intervals are calculated by taking the standard deviation of data in bins perpendicular to the line of best fit, effectively determining the confidence intervals at each point along the line of best fit. In practice, this involves Seaborn bootstrapping the data, a process where new data is created through random sampling with replacement. The number of bootstrapped samples is automatically determined based on the size of the dataset, but can be manually set as well by passing the **n_boot** argument.*

3. Plot the residuals using Seaborn by running the cell containing the following:

```
fig, ax =  
plt.subplots(1, 2)  
  
ax[0] =  
sns.residplot('RM',  
              'MEDV', df, ax=ax[0],  
              scatter_kws={'alpha':  
0.4})  
ax[0].set_ylabel('MDEV  
residuals  $(y - \hat{y})$ ')  
ax[1] =  
sns.residplot('LSTAT',  
              'MEDV', df, ax=ax[1],  
              scatter_kws={'alpha':  
0.4})  
ax[1].set_ylabel('')
```

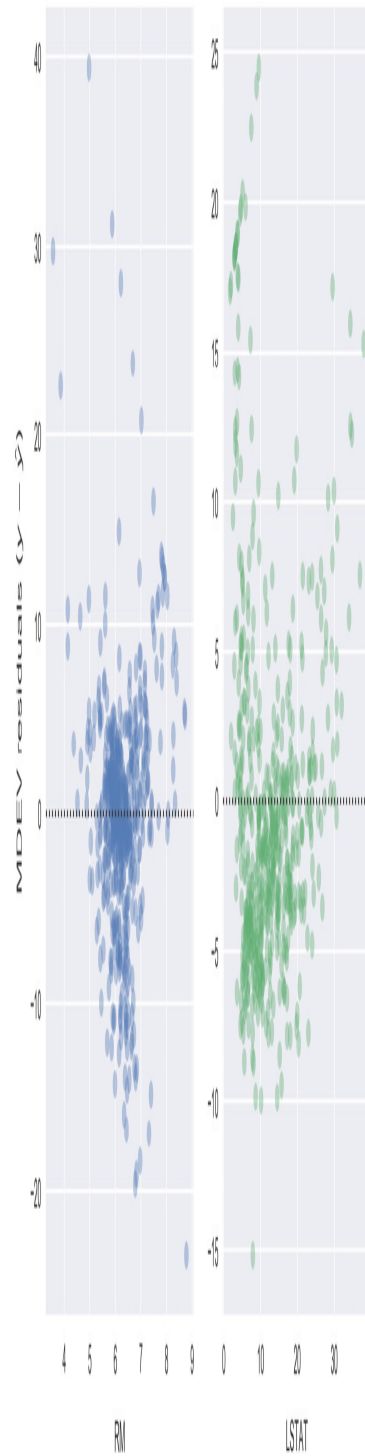


Figure 1.38: Plotting residuals using Seaborn

Each point on these residual plots is

the difference between that sample (y) and the linear model prediction ($\hat{y}$). Residuals greater than zero are data points that would be underestimated by the model.

Likewise, residuals less than zero are data points that would be overestimated by the model.

Patterns in these plots can indicate suboptimal modeling. In each preceding case, we see diagonally arranged scatter points in the positive region. These are caused by the \$50,000 cap on **MEDV**. The **RM** data is clustered nicely around 0, which indicates a good fit. On the other hand, **LSTAT** appears to be clustered lower than 0.

4. Define a function using sci-kit learn that calculates the line of best fit and mean squared error, by running the cell that contains the following:

```
def get_mse(df,
            feature,
            target='MEDV'): # Get
    x, y to model
    y = df[target].values
    x =
```

```
df[feature].values.reshape(-1,1)

...

...

error =
mean_squared_error(y,
y_pred) print('mse =
{:.2f}'.format(error))
print()
```

Note

For complete code, refer to the following: <https://bit.ly/2JgPZdU>

In the **get_mse** function, we first assign the variables **y** and **x** to the target MDEV and the dependent feature, respectively. These are cast as NumPy arrays by calling the **values** attribute. The dependent features array is reshaped to the format expected by scikit-learn; this is only necessary when modeling a one-dimensional feature space. The model is then instantiated and fitted on the data. For linear regression, the fitting consists of computing the model parameters using the ordinary least squares method

(minimizing the sum of squared errors for each sample). Finally, after determining the parameters, we predict the target variable and use the results to calculate the **MSE**.

5. Call the **get_mse** function for both **RM** and **LSTAT**, by running the cell containing the following:

```
get_mse(df, 'RM')  
get_mse(df, 'LSTAT')
```

```
get_mse(df, 'RM')  
get_mse(df, 'LSTAT')
```

MEDV ~ RM

model: $y = -34.671 + 9.102x$

mse = 43.60

MEDV ~ LSTAT

model: $y = 34.554 + -0.950x$

mse = 38.48

Figure 1.39: Calling the get_mse function for RM and LSTAT

Comparing the **MSE**, it turns out the error is slightly lower for **LSTAT**. Looking back to the scatter plots, however, it appears that we might have even better success using a polynomial model for **LSTAT**. In the next activity, we will test this by computing a third-order polynomial model with scikit-learn.

Forgetting about our Boston housing dataset for a minute, consider another real-world situation where you might employ polynomial regression. The following example is modeling weather data. In the following plot, we see temperatures (lines) and precipitations (bars) for Vancouver, BC, Canada:

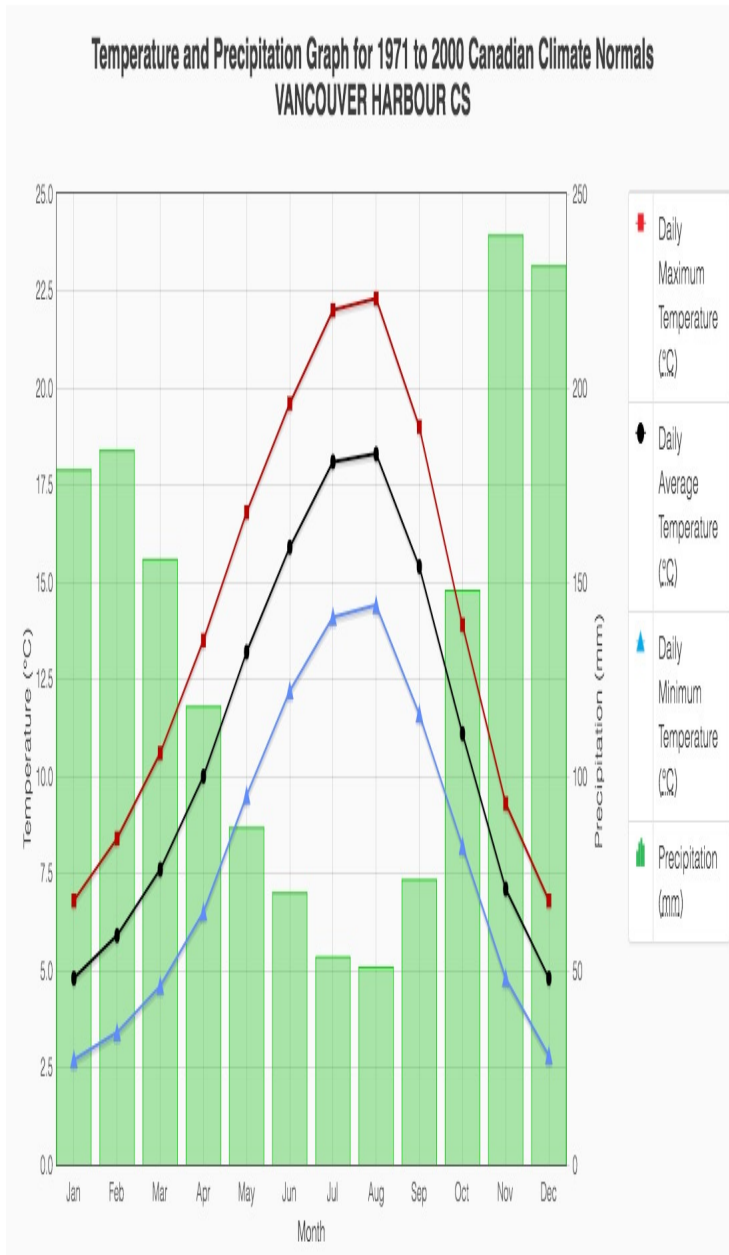


Figure 1.40: Visualizing weather data for Vancouver, Canada

Any of these fields are likely to be fit quite well by a fourth-order polynomial. This would be a very valuable model to have, for example, if you were interested in predicting the

temperature or precipitation for a continuous range of dates.

Note

You can find the data source for this here:

http://climate.weather.gc.ca/climate_normals/results_e.html?stnID=888.

ACTIVITY 1: BUILDING A THIRD-ORDER POLYNOMIAL MODEL

Shifting our attention back to the Boston housing dataset, we would like to build a third-order polynomial model to compare against the linear one. Recall the actual problem we are trying to solve: predicting the median house value, given the lower class population percentage. This model could benefit a prospective Boston house purchaser who cares about how much of their community would be lower class.

Our aim is to use scikit-learn to fit a polynomial regression model to predict the median house value (**MEDV**), given the **LSTAT** values. We are hoping to build a model that has a lower mean-squared error (**MSE**). In order to achieve this, the following steps have to be executed:

1. Scroll to the empty cells at the bottom of **Subtopic C** in your Jupyter Notebook. These will be found beneath the linear-model **MSE** calculation cell under the **Activity** heading.

Note

You should fill these empty cells in with code as we complete the activity. You may need to insert new cells as these become filled up; please do so as needed.

2. Pull out our dependent feature from and target variable from **df**.
3. Verify what **x** looks like by printing the first three samples.
4. Transform **x** into "polynomial features" by importing the appropriate transformation tool from scikit-
5. Transform the **LSTAT** feature (as stored in the variable **x**) by running the **fit_transform** method and build the polynomial feature set.
6. Verify what **x_poly** looks like by printing the first few samples.
7. Import the **LinearRegression** class and build our linear classification model the same way as done while calculating the MSE.
8. Extract the coefficients and print the polynomial model.

9. Determine the predicted values for each sample and calculate the residuals.
10. Print some of the residual values.
11. Print the MSE for the third-order polynomial model.
12. Plot the polynomial model along with the samples.
13. Plot the residuals.

Note

The detailed steps along with the solutions are presented in the Appendix A (pg. no. 144).

Having successfully modeled the data using a polynomial model, let's finish up this chapter by looking at categorical features. In particular, we are going to build a set of categorical features and use them to explore the dataset in more detail.

USING CATEGORICAL FEATURES FOR SEGMENTATION ANALYSIS

Often, we find datasets where there are a mix of continuous and categorical fields. In such cases, we can learn about our

data and find patterns by segmenting the continuous variables with the categorical fields.

As a specific example, imagine you are evaluating the return on investment from an ad campaign. The data you have access to contain measures of some calculated **return on investment (ROI)** metric. These values were calculated and recorded daily and you are analyzing data from the previous year. You have been tasked with finding data-driven insights on ways to improve the ad campaign. Looking at the ROI daily time series, you see a weekly oscillation in the data. Segmenting by day of the week, you find the following ROI distributions (where 0 represents the first day of the week and 6 represents the last).

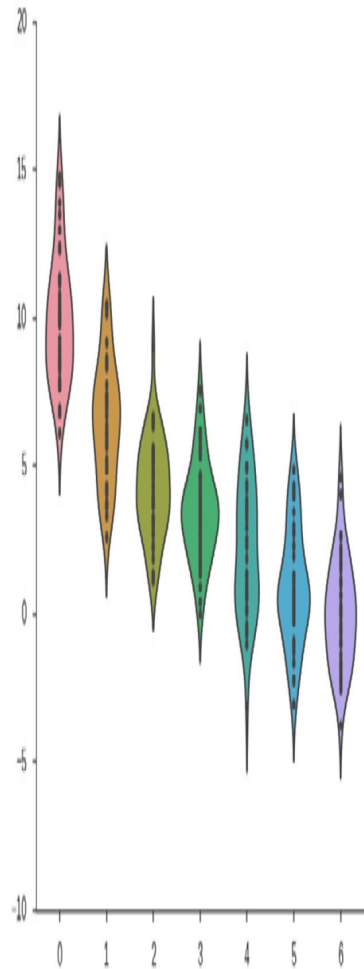


Figure 1.41: A sample violin plot for return on investment

Since we don't have any categorical fields in the Boston housing dataset we are working with, we'll create one by effectively discretizing a continuous field. In our case, this will involve binning the data into "low", "medium", and "high" categories. It's important to note that we are not simply creating a categorical data field to illustrate the data analysis concepts in this section. As will be seen, doing this can reveal insights from the data that would otherwise be difficult to notice or altogether unavailable.

EXERCISE 7: CREATING CATEGORICAL FIELDS FROM CONTINUOUS VARIABLES AND MAKE SEGMENTED VISUALIZATIONS

1. Scroll up to the pairplot in the Jupyter Notebook where we compared **MEDV**, **LSTAT**, **TAX**, **AGE**, and **RM**:

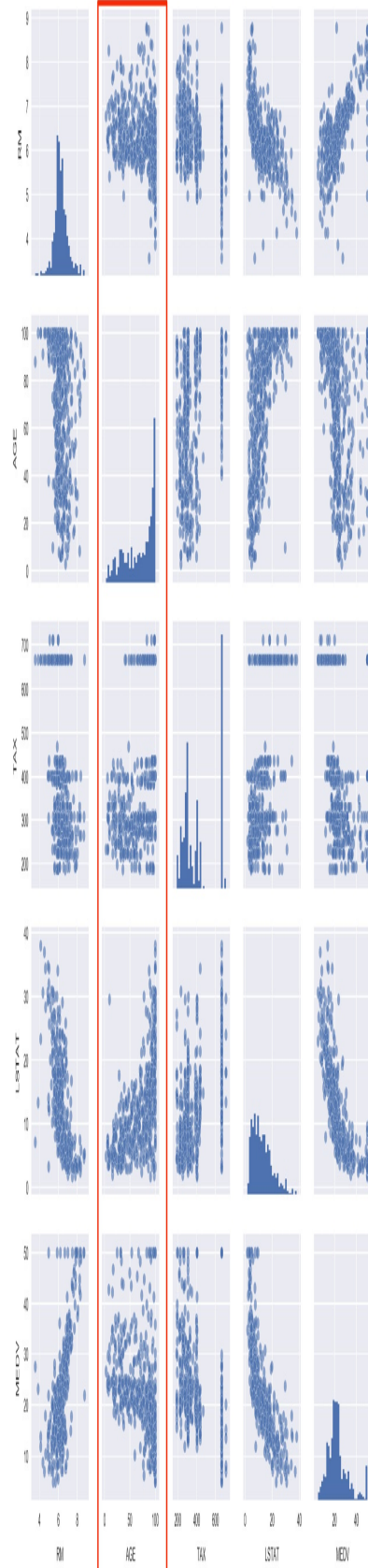


**Figure 1.42: A comparison of plots for
MEDV, LSTAT, TAX, AGE, and RM**

Take a look at the panels containing **AGE**. As a reminder, this feature is defined as the proportion of *owner-occupied units built prior to 1940*. We are going to convert this feature to a categorical variable. Once it's been converted, we'll be able to replot this figure with each panel segmented by color according to the age category.

2. Scroll down to **Subtopic D: Building and exploring categorical features** and click into the first cell. Type and execute the following to plot the **AGE** cumulative distribution:

```
sns.distplot(df.AGE.values, bins=100,
             hist_kws=
             {'cumulative': True},
             kde_kws={'lw': 0})

plt.xlabel('AGE')
plt.ylabel('CDF')
plt.axhline(0.33,
            color='red')
plt.axhline(0.66,
```

```
color='red')  
plt.xlim(0,  
df.AGE.max());
```

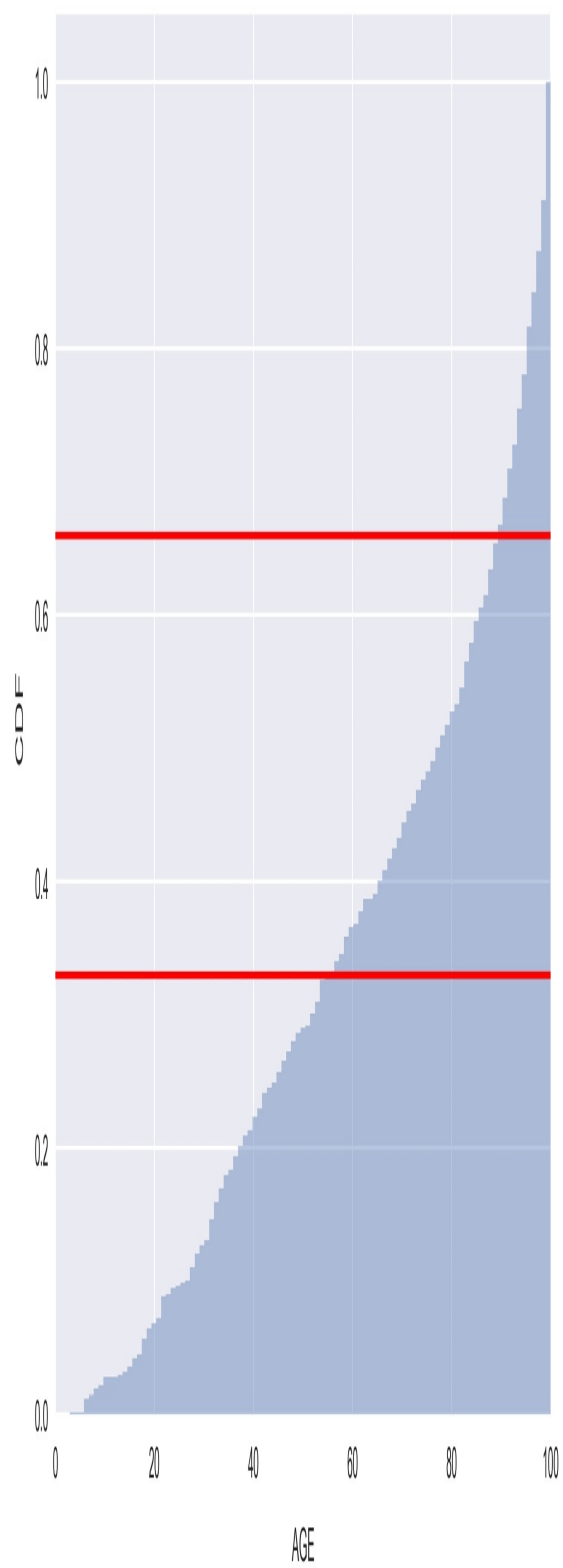


**Figure 1.43: Plot for cumulative
distribution of AGE**

Note that we set **kde_kws=**
{ 'lw' : 0 } in order to bypass
plotting the kernel density estimate
in the preceding figure.

Looking at the plot, there are very
few samples with low **AGE**,
whereas there are far more with a
very large **AGE**. This is indicated
by the steepness of the distribution
on the far right-hand side.

The red lines indicate 1/3 and 2/3
points in the distribution. Looking
at the places where our distribution
intercepts these horizontal lines, we
can see that only about 33% of the
samples have **AGE** less than 55 and
33% of the samples have **AGE**
greater than 90! In other words, a
third of the housing communities
have less than 55% of homes built
prior to 1940. These would be
considered relatively new
communities. On the other end of
the spectrum, another third of the
housing communities have over
90% of homes built prior to 1940.
These would be considered very

old. We'll use the places where the red horizontal lines intercept the distribution as a guide to split the feature into categories: **Relatively New**, **Relatively Old**, and **Very Old**.

3. Create a new categorical feature and set the segmentation points by running the following code:

```
def
get_age_category(x):
    if x < 50:

        return 'Relatively
New' elif 50 <= x <
85:

        return 'Relatively
Old' else:

        return 'Very Old'

df['AGE_category'] =
df.AGE.apply(get_age_c
ategory)
```

Here, we are using the very handy Pandas method `apply`, which applies a function to a given column or set of columns. The function being applied, in this case **`get_age_category`**, should

take one argument representing a row of data and return one value for the new column. In this case, the row of data being passed is just a single value, the **AGE** of the sample.

Note

*The apply method is great because it can solve a variety of problems and allows for easily readable code. Often though, vectorized methods such as **pd.Series.str** can accomplish the same thing much faster. Therefore, it's advised to avoid using it if possible, especially when working with large datasets. We'll see some examples of vectorized methods in the upcoming chapter.*

4. Verify the number of samples we've grouped into each age category by typing **df.groupby('AGE_category').size()** into a new cell and running it:

```
# Check the segmented counts
df.groupby('AGE_category').size()
```

```
AGE_category
Relatively New    147
Relatively Old    149
Very Old          210
dtype: int64
```

Figure 1.44: Verifying the grouping of variables

Looking at the result, it can be seen that two class sizes are fairly equal, and the **Very Old** group is about 40% larger. We are interested in keeping the classes comparable in size, so that each is well-represented and it's straightforward to make inferences from the analysis.

Note

It may not always be possible to assign samples into classes evenly, and in real-world situations, it's very common to find highly imbalanced classes. In such cases, it's important to keep in mind that it

will be difficult to make statistically significant claims with respect to the under-represented class.

Predictive analytics with imbalanced classes can be particularly difficult. The following blog post offers an excellent summary on methods for handling imbalanced classes when doing machine learning:

<https://svds.com/learning-imbalanced-classes/>.

Let's see how the target variable is distributed when segmented by our new feature **AGE_category**.

5. Construct a violin plot by running the following code:

```
sns.violinplot(x='MEDV', y='AGE_category', data=df, order=['Relatively New', 'Relatively Old', 'Very Old']);
```

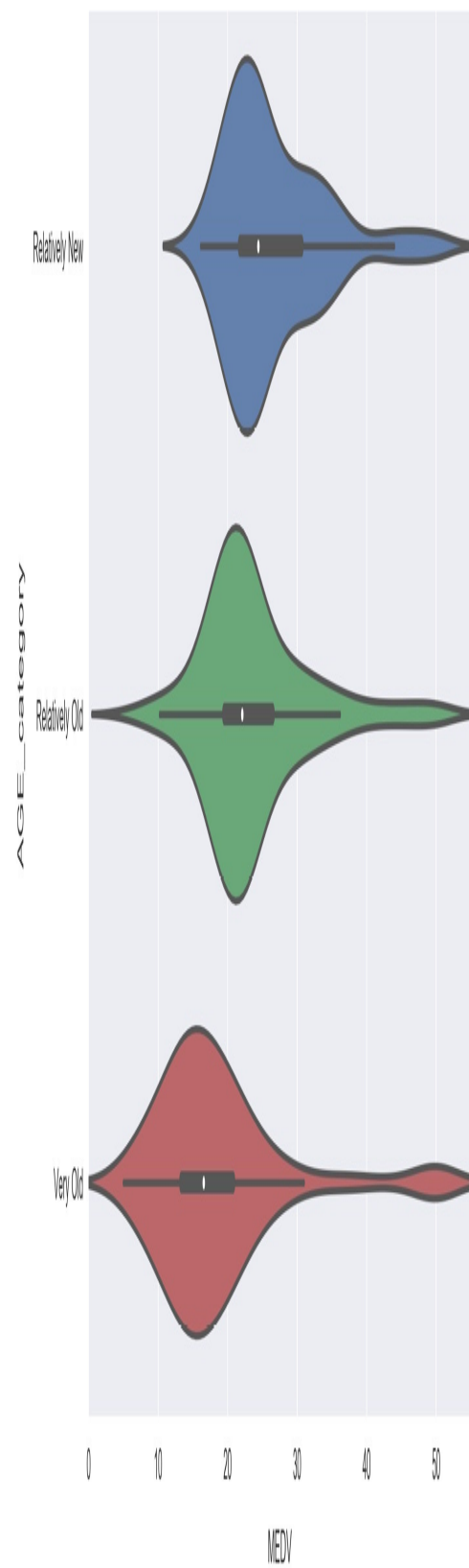


**Figure 1.45: Violin plot for
AGE_category and MEDV**

The violin plot shows a kernel density estimate of the median house value distribution for each age category. We see that they all resemble a normal distribution. The Very Old group contains the lowest median house value samples and has a relatively large width, whereas the other groups are more tightly centered around their average. The young group is skewed to the high end, which is evident from the enlarged right half and position of the white dot in the thick black line within the body of the distribution.

This white dot represents the mean and the thick black line spans roughly 50% of the population (it fills to the first quantile on either side of the white dot). The thin black line represents boxplot whiskers and spans 95% of the population. This inner visualization can be modified to show the individual data points instead by passing **inner='point'** to

sns.violinplot(). Let's do that now.

6. Re-construct the violin plot adding the **inner='point'** argument to the **sns.violinplot** call:

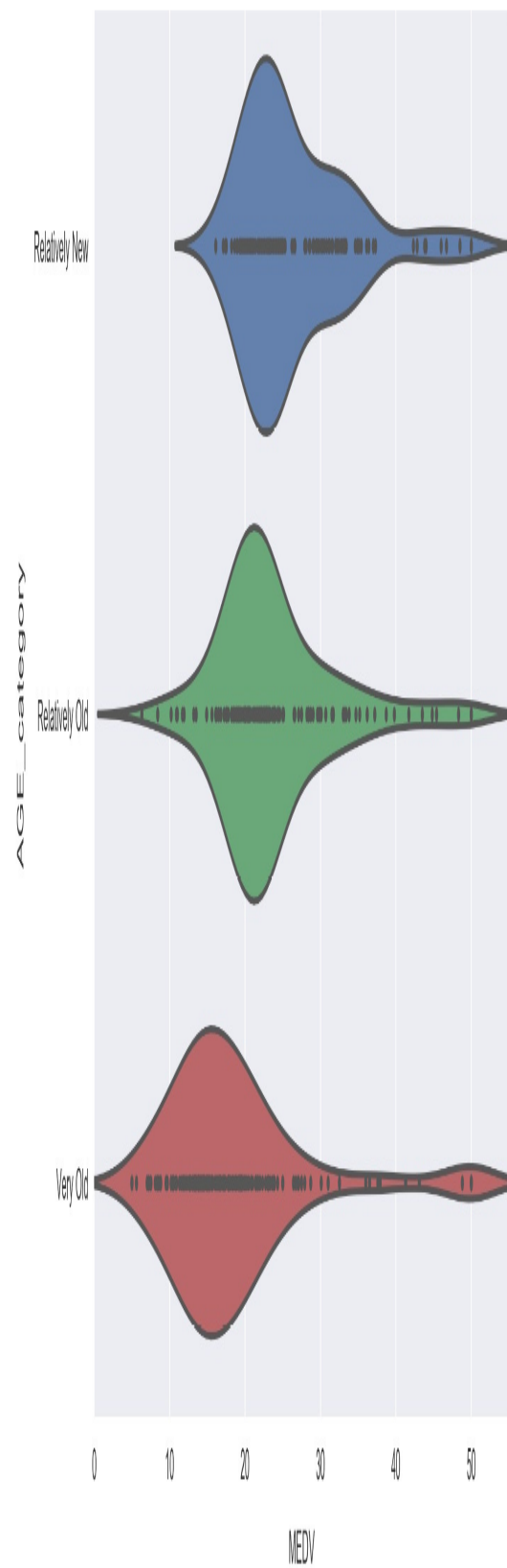


Figure 1.46: Violin plot for AGE_category and MEDV with the inner = 'point' argument

It's good to make plots like this for test purposes in order to see how the underlying data connects to the visual. We can see, for example, how there are no median house values lower than roughly \$16,000 for the **Relatively New** segment, and therefore the distribution tail actually contains no data. Due to the small size of our dataset (only about 500 rows), we can see this is the case for each segment.

7. Re-construct the pairplot from earlier, but now include color labels for each **AGE** category. This is done by simply passing the **hue** argument, as follows:

```
cols = ['RM', 'AGE',  
        'TAX', 'LSTAT',  
        'MEDV', 'AGE_  
category']  
  
sns.pairplot(df[cols],  
             hue='AGE_category',  
             hue_order=['Relatively  
New', 'Relatively
```

```
old',  
    'Very Old'],  
    plot_kws={'alpha':  
0.5}, diag_kws=  
{'bins':  
30});
```

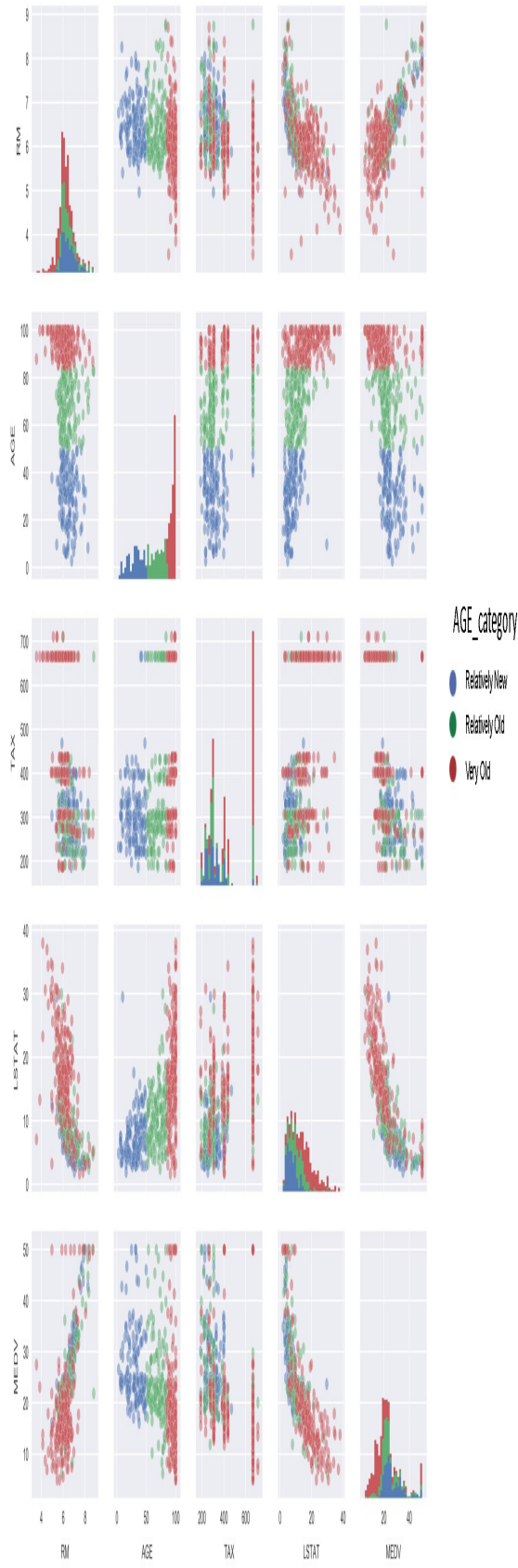


Figure 1.47: Re-constructing pairplot for all variables using color labels for AGE

Looking at the histograms, the underlying distributions of each segment appear similar for **RM** and **TAX**. The **LSTAT** distributions, on the other hand, look more distinct. We can focus on them in more detail by again using a violin plot.

8. Re-construct a violin plot comparing the LSTAT distributions for each **AGE_category** segment:

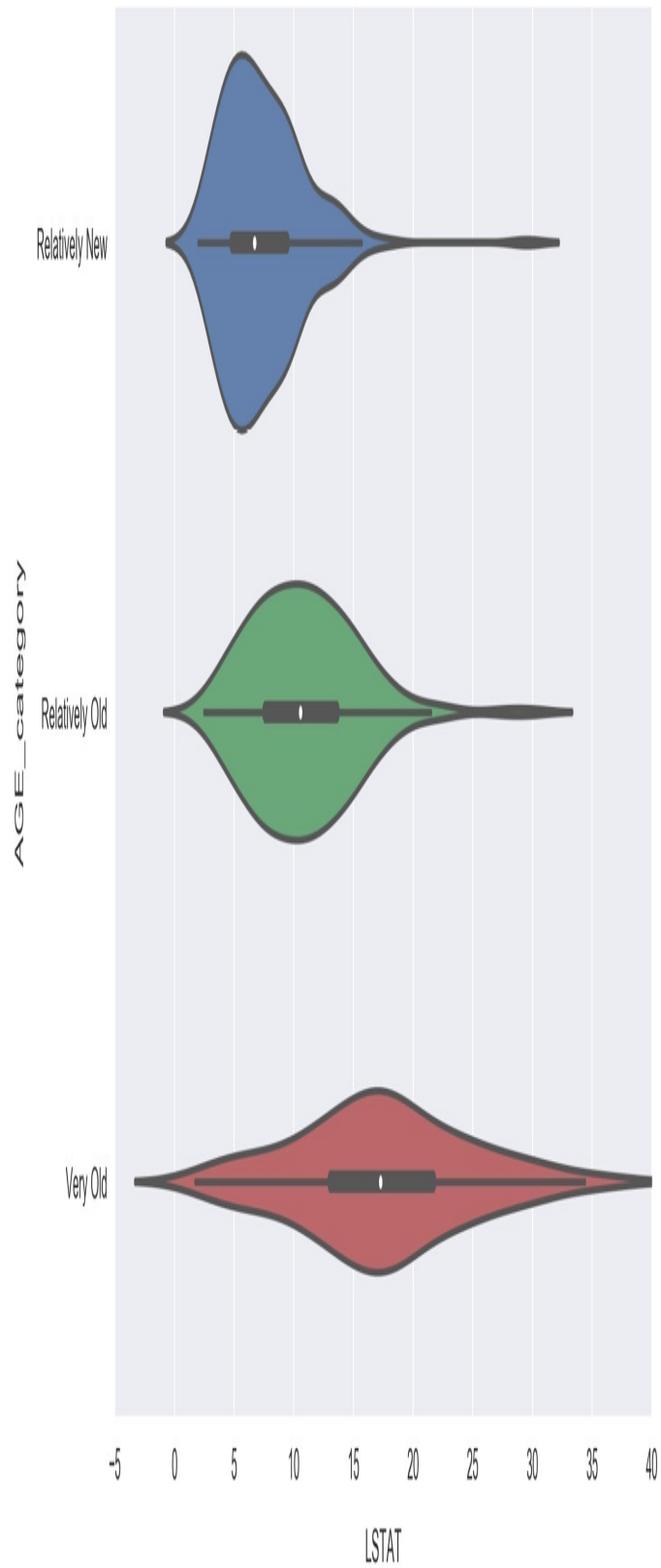


Figure 1.48: Re-constructed violin plots for

comparing LSTAT distributions for the AGE_category

Unlike the **MEDV** violin plot, where each distribution had roughly the same width, here we see the width increasing along with **AGE**. Communities with primarily old houses (the **Very Old** segment) contain anywhere from very few to many lower class residents, whereas **Relatively New** communities are much more likely to be predominantly higher class, with over 95% of samples having less lower class percentages than the **Very Old** communities. This makes sense, because **Relatively New** neighborhoods would be more expensive.

Summary

In this chapter, you have seen the fundamentals of data analysis in Jupyter. We began with usage instructions and features of Jupyter such as magic functions and tab completion. Then, transitioning to data-science-specific material, we introduced the most important libraries for data science with Python.

In the latter half of the chapter, we ran an exploratory analysis in a live Jupyter Notebook. Here, we used visual assists such as scatter plots, histograms, and violin plots to deepen our understanding of the data. We also performed simple predictive modeling, a topic which will be the focus of the following chapter in this book.

In the next chapter, we will discuss how to approach predictive analytics, what things to consider when preparing the data for modeling, and how to implement and compare a variety of models using Jupyter Notebooks.

Data Cleaning and Advanced Machine Learning

Learning Objectives

By the end of this chapter, you will be able to:

- Plan a machine learning classification strategy
- Preprocess data to prepare it for machine learning
- Train classification models
- Use validation curves to tune model parameters
- Use dimensionality reduction to enhance model performance

In this chapter, you will learn data preprocessing and machine learning by completing several practical exercises.

Introduction

Consider a small food-delivery business that is looking to optimize their product. An analyst might look at the appropriate data and determine what type of food people are

enjoying most. Perhaps they find a large amount of people are ordering the spiciest food options, ...

Web Scraping and Interactive Visualizations

Learning Objectives

By the end of this chapter, you will be able to:

- Describe how HTTP requests work
- Scrape tabular data from a web page
- Build and transform Pandas DataFrames
- Create interactive visualizations

In this chapter, you will learn the fundamentals of HTTP requests, scrape web page data, and then create interactive visualizations using the Jupyter Notebook.

Introduction

So far in this book, we have focused on using Jupyter to build reproducible data analysis pipelines and predictive models. We'll continue to explore these topics in this chapter, but the main focus here is data acquisition. In particular, we will show you how data can be acquired from the web using HTTP requests. This will involve scraping web pages by requesting

and parsing HTML. We will then wrap up this chapter by using interactive visualization techniques to explore the data we've collected.

The amount of data available online is huge and relatively easy to acquire. It's also continuously growing and becoming increasingly important. Part of this continual growth is the result of an ongoing global shift from newspapers, magazines, and TV to online content. With customized newsfeeds available all the time on cell phones, and live-news sources such as Facebook, Reddit, Twitter, and YouTube, it's difficult to imagine the historical alternatives being relevant much longer. Amazingly, this accounts for only some of the increasingly massive amounts of data available online.

With this global shift toward consuming content using HTTP services (blogs, news sites, Netflix, and so on), there are plenty of opportunities to use data-driven analytics. For example, Netflix looks at the movies a user watches and predicts what they will like. This prediction is used to determine the suggested movies that appear. In this chapter, however, we won't be looking at "business-facing" data as such, but instead we will see how the client can leverage the internet as a database. Never before has this amount and variety of data been so easily accessible. We'll use web-scraping techniques to collect data, and then we'll explore it with interactive visualizations in Jupyter.

Interactive visualization is a visual form of data representation, which helps users understand the data using graphs or charts. Interactive visualization helps a developer or analyst present data in a simple form, which can be understood by non-technical personnel too.

Scraping Web Page Data

In the spirit of leveraging the internet as a database, we can

think about acquiring data from web pages either by scraping content or by interfacing with web APIs. Generally, scraping content means getting the computer to read data that was intended to be displayed in a human-readable format. This is in contradistinction to web APIs, where data is delivered in machine-readable formats—the most common being JSON.

In this topic, we will focus on web scraping. The exact process for doing this will depend on the page and desired content. However, as we will see, it's quite easy to scrape anything we need from an HTML page so long as we have an understanding of the underlying concepts and tools. In this topic, we'll use Wikipedia as an example and scrape tabular content from an article. Then, we'll apply the same techniques to scrape

data from a page on an entirely separate domain. But first, we'll take some time to introduce HTTP requests.

INTRODUCTION TO HTTP REQUESTS

The Hypertext Transfer Protocol, or HTTP for short, is the foundation of data communication for the internet. It defines how a page should be requested and how the response should look. For example, a client can request an Amazon page of laptops for sale, a Google search of local restaurants, or their Facebook feed. Along with the URL, the request will contain the user agent and available browsing cookies among the contents of the request header. The user agent tells the server what browser and device the client is using, which is usually used to provide the most user-friendly version of the web page's response. Perhaps they have recently logged in to the web page; such information would be stored in a cookie that might be used to automatically log the user in.

These details of HTTP requests and responses are taken care of under the hood thanks to web browsers. Luckily for us, today the same is true when making requests with high-level languages such as Python. For many purposes, the contents of

request headers can be largely ignored. Unless otherwise specified, these are automatically generated in Python when requesting a URL. Still, for the purposes of troubleshooting and understanding the responses yielded by our requests, it's useful to have a foundational understanding of HTTP.

There are many types of HTTP methods, such as GET, HEAD, POST, and PUT. The first two are used for requesting that data be sent from the server to the client, whereas the last two are used for sending data to the server.

Note

*Take a look at this GET request example for the **Profile** page on the site <https://www.studytonight.com/>. The exact page that was requested contains parameters, which start after the question mark (?) and are separated by the ampersands (&). These are usually used to modify the page specified by the path to source on the web server. In this case, the **User-Agent** is Mozilla/5.0, which corresponds to a standard desktop browser. Among other lines in the header, we note the **Accept** and **Accept-Language** fields, which specify the acceptable content types and language of the response.*

These HTTP methods are summarized below:

- **GET:** Retrieves the information from the specified URL
- **HEAD:** Retrieves the meta information from the HTTP header of the specified URL
- **POST:** Sends the attached information for appending to the resource(s) at the specified URL
- **PUT:** Sends the attached

information for replacing the resource(s) at the specified URL

A **GET** request is sent each time we type a web page address into our browser and press Enter. For web scraping, this is usually the only HTTP method we are interested in, and it's the only method we'll be using in this chapter.

Once the request has been sent, a variety of response types can be returned from the server. These are labeled with 100-level to 500-level codes, where the first digit in the code represents the response class. These can be described as follows:

- **1xx**: Informational response, for example, server is processing a request. It's uncommon to see this.
- **2xx**: Success, for example, page has loaded properly.
- **3xx**: Redirection, for example, the requested resource has been moved and we were redirected to a new URL.
- **4xx**: Client error, for example, the requested resource does not exist.
- **5xx**: Server error, for example, the website server is receiving too much traffic and could not fulfill the request.

For the purposes of web scraping, we usually only care about the response class, that is, the first digit of the response code.

However, there exist subcategories of responses within each class that offer more granularity on what's going on. For example, a 401 code indicates an unauthorized response, whereas a 404 code indicates a page not found response. This distinction is noteworthy because a 404 would indicate we've requested a page that does not exist, whereas 401 tells us we need to log in to view the particular resource.

Let's see how HTTP requests can be done in Python and explore some of these topics using the Jupyter Notebook.

MAKING HTTP REQUESTS IN THE JUPYTER NOTEBOOK

Now that we've talked about how HTTP requests work and what type of responses we should expect, let's see how this can be done in Python. We'll use a library called **Requests**, which happens to be the most downloaded external library for Python. It's possible to use Python's built-in tools, such as **urllib**, for making HTTP requests, but **Requests** is far more intuitive, and in fact it's recommended over **urllib** in the official Python documentation.

Requests is a great choice for making simple and advanced web requests. It allows for all sorts of customization with respect to headers, cookies, and authorization. It tracks redirects and provides methods for returning specific page content such as JSON. Furthermore, there's an extensive suite of advanced features. However, it does not allow JavaScript to be rendered.

Oftentimes, servers return HTML with JavaScript code snippets included, which are automatically run in the browser on load time. When requesting content with Python using Requests, this JavaScript code is visible, but it does not run. Therefore, any elements that would be altered or created by doing so are missing. Often, this does not affect the ability to

get the desired information, but in some cases we may need to render the JavaScript in order to scrape the page properly. For doing this, we could use a library like Selenium.

This has a similar API to the Requests library, but provides support for rendering JavaScript using web drivers. It can even run JavaScript commands on live pages, for example, to change the text color or scroll to the bottom of the page.

Note

For more information, refer to: <http://docs.python-requests.org/en/master/user/advanced/> and <http://selenium-python.readthedocs.io/>.

Let's dive into an exercise using the Requests library with Python in a Jupyter Notebook.

EXERCISE 14: HANDLING HTTP REQUESTS WITH PYTHON IN A JUPYTER NOTEBOOK

1. Start the **NotebookApp** from the project directory by executing `jupyter notebook`. Navigate to the **lesson-3** directory and open up the **lesson-3-workbook.ipynb** file. Find the cell near the top where the packages are loaded and run it.

We are going to request a web page

and then examine the response object. There are many different libraries for making requests and many choices for exactly how to do so with each. We'll only use the Requests library, as it provides excellent documentation, advanced features, and a simple API.

2. Scroll down to **Subtopic A: Introduction to HTTP requests** and run the first cell in that section to import the Requests library. Then, prepare a request by running the cell containing the following code:

```
url =  
'https://jupyter.org/'  
  
req =  
requests.Request('GET'  
, url)  
req.headers['User-  
Agent'] =  
'Mozilla/5.0'  
  
req = req.prepare()
```

We use the Request class to prepare a GET request to the jupyter.org homepage. By specifying the user agent as Mozilla/5.0, we are asking

for a response that would be suitable for a standard desktop browser. Finally, we prepare the request.

3. Print the docstring for the "prepared request" req, by running the cell containing **req?**:

```
In [83]: req?
```

```
Type: PreparedRequest
String form: <PreparedRequest [GET]>
File: ~/anaconda/lib/python3.5/site-packages/requests/models.py
Docstring:
The fully mutable :class:`PreparedRequest` <PreparedRequest> object,
containing the exact bytes that will be sent to the server.

Generated from either a :class:`Request` <Request> object or manually.

Usage::

>>> import requests
>>> req = requests.Request('GET', 'http://httpbin.org/get')
>>> r = req.prepare()
<PreparedRequest [GET]>

>>> s = requests.Session()
>>> s.send(r)
<Response [200]>
```

Figure 3.1: Printing the docstring for req

Looking at its usage, we see how the request can be sent using a session. This is similar to opening a web browser (starting a session) and then requesting a URL.

4. Make the request and store the

response in a variable named `page`, by running the following code:

```
with  
requests.Session() as  
sess: page =  
sess.send(req)
```

This code returns the HTTP response, as referenced by the `page` variable. By using the **`with`** statement, we initialize a session whose scope is limited to the indented code block. This means we do not have to worry about explicitly closing the session, as it is done automatically.

5. Run the next two cells in the notebook to investigate the response. The string representation of `page` should indicate a 200 status code response. This should agree with the **`status_code`** attribute.
6. Save the response text to the **`page_html`** variable and take a look at the head of the string with **`page_html[:1000]`**:

```
page_html = page.text
```

```
page_html[:1000]
```

```
<!DOCTYPE html>\n<html>\n\n  <head>\n\n    <meta charset="utf-8">\n    <meta http-equiv="X-UA-  
Compatible" content="IE=edge">\n    <meta name="viewport" content="width=device-width, init  
ial-scale=1">\n    <meta name="description" content="">\n    <meta name="author" content="">\n\n    <title>Project Jupyter | Home</title>\n    <meta property="og:title" content="Project  
Jupyter" />\n    <meta property="og:description" content="The Jupyter Notebook is a web-based  
interactive computing platform. The notebook combines live code, equations, narrative text, v  
isualizations, interactive dashboards and other media.\n">\n    <meta property="og:url" conte  
nt="http://www.jupyter.org" />\n    <meta property="og:image" content="http://jupyter.org/ass  
ets/homepage.png" />\n    <!-- Bootstrap Core CSS -->\n    <script src="/cdn-cgi/apps/head/ku  
I14I_IYFKldaVulndWeeYas.js"></script><link rel="stylesheet" href="/css/bootstrap.min.css">\n    <link rel="stylesheet" href="/css/logo-nav.css">\n    <link rel="stylesheet" href="/c'
```

Figure 3.2: The HTML response text

As expected, the response is HTML. We can format this output better with the help of **BeautifulSoup**, a library which will be used extensively for HTML parsing later in this section.

7. Print the head of the formatted HTML by running the following:

```
from bs4 import  
BeautifulSoup  
  
print(BeautifulSoup(pa  
ge_html,  
'html.parser').prettif  
y() [:1000])
```

We import **BeautifulSoup** and then print the output, where newlines are indented depending on

their hierarchy in the HTML structure.

8. We can take this a step further and actually display the HTML in Jupyter by using the IPython display module. Do this by running the following code:

```
from IPython.display
import HTML
HTML(page_html)Here,
we see the HTML
rendered as well as
possible, given that
no JavaScript code has
been run and no
external resources
have loaded. For
example, the images
that are hosted on the
jupyter.org server are
not rendered and we
instead see the alt
text: circle of
programming icons,
Jupyter logo, and so
on.
```

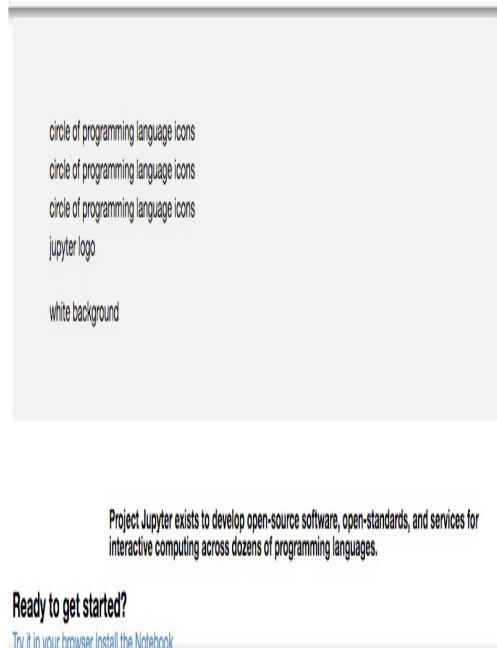


Figure 3.3: The output obtained when no images are loaded

9. Let's compare this to the live website, which can be opened in Jupyter using an IFrame. Do this by running the following code:

```
from IPython.display
import IFrame
IFrame(src=url,
height=800, width=800)
```



[Install](#) [About Us](#) [Community](#) [Documentation](#) [NBViewer](#) [Widgets](#) [Blog](#)



Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages.

Figure 3.4: Rendering of the entire Jupyter website

Here, we see the full site rendered, including JavaScript and external resources. In fact, we can even click on the hyperlinks and load those pages in the IFrame, just like a regular browsing session.

10. It's good practice to close the `IFrame` after using it. This prevents it from eating up memory and processing power. It can be closed by selecting the cell and clicking **Current Outputs | Clear** from the Cell menu in the Jupyter Notebook.

Recall how we used a prepared request and session to request this content as a string in Python. This is often done using a shorthand method instead. The drawback is that we do not have as much customization of the request header, but that's usually fine.

11. Make a request to <http://www.python.org/> by running the following code:

```
url =  
'http://www.python.org'  
' page =  
requests.get(url)  
  
page  
  
<Response [200]>
```

The string representation of the page (as displayed beneath the cell) should indicate a 200 status code,

indicating a successful response.

12. Run the next two cells. Here, we print the **url** and history attributes of our page.

The URL returned is not what we input; notice the difference? We were redirected from the input URL, <http://www.python.org/>, to the secured version of that page, <https://www.python.org/>. The difference is indicated by an additional s at the start of the URL, in the protocol. Any redirects are stored in the history attribute; in this case, we find one page in here with status code 301 (permanent redirect), corresponding to the original URL requested.

Now that we're comfortable making requests, we'll turn our attention to parsing the HTML. This can be something of an art, as there are usually multiple ways to approach it, and the best method often depends on the details of the specific HTML in question.

PARSING HTML IN THE JUPYTER NOTEBOOK

When scraping data from a web page, after making the request, we must extract the data from the response content. If the

content is HTML, then the easiest way to do this is with a high-level parsing library such as BeautifulSoup. This is not to say it's the only way; in principle, it would be possible to pick out the data using regular expressions or Python string methods such as split, but pursuing either of these options would be an inefficient use of time and could easily lead to errors. Therefore, it's generally frowned upon and instead, the use of a trustworthy parsing tool is recommended.

In order to understand how content can be extracted from HTML, it's important to know the fundamentals of HTML. For starters, HTML stands for Hyper Text Markup Language. Like Markdown or XML (eXtensible Markup Language), it's simply a language for marking up text. In HTML, the display text is contained within the content section of HTML elements, where element attributes specify how that element should appear on the page.



Figure 3.5: Fundamental blocks of HTML

Looking at the anatomy of an HTML element, as seen in the preceding picture, we see the content enclosed between start and end tags. In this example, the tags are **<p>** for paragraph; other common tag types are **<div>** (text block), **<table>** (data table),

<h1> (heading), **** (image), and **<a>** (hyperlinks). Tags have attributes, which can hold important metadata. Most commonly, this metadata is used to specify how the element text should appear on the page. This is where CSS files come into play. The attributes can store other useful information, such as the hyperlink **href** in an **<a>** tag, which specifies a URL link, or the alternate alt label in an **** tag, which

specifies the text to display if the image resource cannot be loaded.

Now, let's turn our attention back to the Jupyter Notebook and parse some HTML! Although not necessary when following along with this exercise, it's very helpful in real-world situations to use the developer tools in Chrome or Firefox to help identify the HTML elements of interest. We'll include instructions for doing this with Chrome in the following exercise.

EXERCISE 15: PARSING HTML WITH PYTHON IN A JUPYTER NOTEBOOK

1. In **lesson-3-workbook.ipynb** file, scroll to the top of **Subtopic B: Parsing HTML** with Python.

In this exercise, we'll scrape the central bank interest rates for each country, as reported by Wikipedia. Before diving into the code, let's first open up the web page containing this data.

2. Go to https://en.wikipedia.org/wiki/List_of_countries_by_central_bank_interest_rates in a web browser. Use Chrome, if possible, as later in this

exercise we'll show you how to view and search the HTML using Chrome's developer tools.

Looking at the page, we see very little content other than a big list of countries and their interest rates. This is the table we'll be scraping.

3. Return to the Jupyter Notebook and load the HTML as a BeautifulSoup object so that it can be parsed. Do this by running the following code:

```
from bs4 import
BeautifulSoup

soup =
BeautifulSoup(page.con
tent, 'html.parser')
```

We use Python's default `html.parser` as the parser, but third-party parsers such as **lxml** may be used instead, if desired. Usually, when working with a new object like this BeautifulSoup one, it's a good idea to pull up the docstring by doing **soup?**. However, in this case, the docstring is not particularly informative. Another tool for exploring Python objects is **pdir**,

which lists all of an object's attributes and methods (this can be installed with `pip install pdir2`). It's basically a formatted version of Python's built-in `dir` function.

4. Display the attributes and methods for the BeautifulSoup object by running the following code. This will run, regardless of whether or not the `pdir` external library is installed:

```
try:
    import pdir
    dir = pdir
except:
    print('You can install
    pdir with:\npip
    install pdir2')
    dir(soup)
```

Here, we see a list of methods and attributes that can be called on soup. The most commonly used function is probably `find_all`, which returns a list of elements that match the given criteria.

5. Get the h1 heading for the page with the following code:

```
h1 =  
soup.find_all('h1') h1  
  
>> [<h1  
class="firstHeading"  
id="firstHeading"  
lang="en">List of  
countries by central  
bank interest  
rates</h1>]
```

Usually, pages only have one H1 (top-level heading) element, so it's no surprise that we only find one here.

6. Run the next couple of cells. We redefine H1 to the first (and only) list element with **h1 = h1[0]**, and then print out the HTML element attributes with **h1.attrs**:

We see the class and ID of this element, which can both be referenced by CSS code to define the style of this element.

7. Get the HTML element content (that is, the visible text) by printing **h1.text**.

8. Get all the images on the page by running the following code:

```
imgs =  
soup.find_all('img')  
len(imgs)  
  
>> 91
```

There are lots of images on the page. Most of these are for the country flags.

9. Print the source of each image by running the following code:

```
[element.attrs['src']  
for element in imgs if  
'src' in  
element.attrs.keys()]
```

We use a list comprehension to iterate through the elements, selecting the **src** attribute of each (so long as that attribute is actually available).

Now, let's scrape the table. We'll use Chrome's developer tools to hunt down the element this is contained within.

```
'//upload.wikimedia.org/wikipedia/commons/thumb/3/36/Flag_of_Albania.svg/21px-Flag_of_Albania.svg.png',  
'//upload.wikimedia.org/wikipedia/commons/thumb/9/9d/Flag_of_Angola.svg/23px-Flag_of_Angola.svg.png',  
'//upload.wikimedia.org/wikipedia/commons/thumb/1/1a/Flag_of_Argentina.svg/23px-Flag_of_Argentina.svg.png',  
'//upload.wikimedia.org/wikipedia/commons/thumb/2/2f/Flag_of_Armenia.svg/23px-Flag_of_Armenia.svg.png',  
'//upload.wikimedia.org/wikipedia/en/thumb/b/b9/Flag_of_Australia.svg/23px-Flag_of_Australia.svg.png',  
'//upload.wikimedia.org/wikipedia/commons/thumb/d/dd/Flag_of_Azerbaijan.svg/23px-Flag_of_Azerbaijan.svg.png',  
'//upload.wikimedia.org/wikipedia/commons/thumb/9/93/Flag_of_the_Bahamas.svg/23px-Flag_of_the_Bahamas.svg.png',  
'//upload.wikimedia.org/wikipedia/commons/thumb/2/2c/Flag_of_Bahrain.svg/23px-Flag_of_Bahrain.svg.png',  
'//upload.wikimedia.org/wikipedia/commons/thumb/f/f9/Flag_of_Bangladesh.svg/23px-Flag_of_Bangladesh.svg.png',  
'//upload.wikimedia.org/wikipedia/commons/thumb/e/ef/Flag_of_Barbados.svg/23px-Flag_of_Barbados.svg.png',  
'//upload.wikimedia.org/wikipedia/commons/thumb/8/85/Flag_of_Belarus.svg/23px-Flag_of_Belarus.svg.png',  
'//upload.wikimedia.org/wikipedia/commons/thumb/f/fa/Flag_of_Botswana.svg/23px-Flag_of_Botswana.svg.png',  
'//upload.wikimedia.org/wikipedia/en/thumb/0/05/Flag_of_Brazil.svg/22px-Flag_of_Brazil.svg.png',  
'//upload.wikimedia.org/wikipedia/commons/thumb/9/9a/Flag_of_Bulgaria.svg/23px-Flag_of_Bulgaria.svg.png',
```

Figure 3.6: Scraping the table on the target web page

10. If not already done, open the Wikipedia page we're looking at in Chrome. Then, in the browser, select **Developer Tools** from the **View** menu. A sidebar will open. The HTML is available to look at from the **Elements** tab in Developer Tools.
11. Select the little arrow in the top left of the tools sidebar. This allows us to hover over the page and see where the HTML element is located, in the **Elements** section of the sidebar:

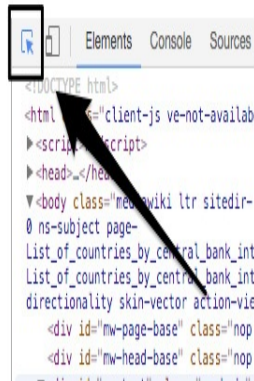


Figure 3.7: Arrow Icon for locating the HTML element

12. Hover over the body to see how the table is contained within the div that has **id="bodyContent"**:

divBodyContent.ms-body-content 447x3553-33

This is a list of countries by annualized interest rate set by the central bank for charging commercial, depository banks for loans to meet temporary shortages of funds.

List [[edit](#)]

Country or currency union	Central bank interest rate (%)	Date of last change
 Albania	1.25	4 May 2016 ^[1]
 Angola	16.00	30 June 2016 ^[1]
 Argentina	26.25	11 April 2017 ^[1]
 Armenia	6.00	14 February 2017 ^[1]
 Australia	1.50	2 August 2016 ^[1]
 Azerbaijan	15.00	9 September 2016 ^[1]
 Bahamas	4.00	22 December 2016 ^[1]
 Bahrain	1.50	11 June 2016 ^[1]

```

<DOCTYPE html>
<html class="client-js ie-not-available lang-en dir-ltr">
<script></script>
<head></head>

<body class="no-js (tr-sitedir-ltr m-body-copy-0) no-js no-subset-page">
  List of countries by central bank interest rates mortgage-
  List of countries by central bank interest rates vector-m-
  dimensionality size-vector action-size = 51

  <div id="m-page-base" class="logprint"></div>
  <div id="m-head-base" class="logprint"></div>
  <div id="content" class="m-body" role="main">
    <a id="top"></a>
    <div id="siteHeader" class="m-body-content"></div>
    <div class="m-indicators m-body-content">
      </div>
    <div id="firstHeading" class="firstHeading" lang="en">
      </div>
    <div id="topContent" class="m-body-content"></div>
    </div>
    <div id="m-adaptation"></div>
    <div id="footer" role="contentinfo"></div>
  <script></script>
  <script></script>

  <a accesskey="f" href="https://en.wikipedia.org/wiki/List_of_countries_by_central_bank_interest_rates"
  title="List of countries by central bank interest rates"
  id="topContent" style="display: none; font-size: 10px;"></div>
  <div id="m-page-wrap"></div>
</body>
</html>

```

Figure 3.8: HTML code for table on the target web page

13. Select that **div** by running the following code:

```
body_content =  
soup.find('div',  
{ 'id': 'bodyContent' })
```

We can now seek out the table within this subset of the full HTML. Usually, tables are organized into headers **<th>**, rows **<tr>**, and data entries **<td>**.

14. Get the table headers by running the following code:

```
table_headers =  
body_content.find_all(  
'th')[:3]  
table_headers  
  
>>> [<th>Country  
or<br/>  
  
currency union</th>,  
<th>Central bank<br/>  
interest rate (%)  
</th>, <th>Date of  
last<br/> change</th>]
```

Here, we see three headers. In the

content of each is a break element **
**, which will make the text a bit more difficult to cleanly parse.

15. Get the text by running the following code:

```
table_headers =  
[element.get_text().re  
place('\n', ' ')]  
  
for element in  
table_headers]  
  
table_headers  
  
>> ['Country or  
currency union',  
'Central bank interest  
rate (%)', 'Date of  
last change']
```

Here, we get the content with the **get_text** method, and then run the replace string method to remove the newline resulting from the **
** element.

To get the data, we'll first perform some tests and then scrape all the data in a single cell.

16. Get the data for each cell in the second **<tr>** (row) element by

running the following code:

```
row_number = 2

d1, d2, d3 =
body_content.find_all(
'tr')[row_number]\

.find_all('td')
```

We find all the row elements, pick out the third one, and then find the three data elements inside that.

Let's look at the resulting data and see how to parse the text from each row.

17. Run the next couple of cells to print **d1** and its text attribute:

```
d1

<td align="left"><span class="flagicon"> </span><a href="/wiki/Angola" title="Angola">Angola</a></td>

d1.text

'\xa0Angola'
```

Figure 3.9: Printing d1 and its text attribute

We're getting some undesirable characters at the front. This can be

solved by searching for only the text of the **<a>** tag.

18. Run **d1.find('a').text** to return the properly *cleaned* data for that cell.
19. Run the next couple of cells to print **d2** and its text. This data appears to be clean enough to convert directly into a float.
20. Run the next couple of cells to print **d3** and its text:

```
d3
<td><span class="sortkey" style="display:none; speak:none">000000002016-06-30-0000</span><span
style="white-space:nowrap">30 June 2016</span><sup class="reference" id="cite_ref-CentralBank
News_1-1"><a href="#cite_note-CentralBankNews-1">[1]</a></sup></td>

d3.text
'000000002016-06-30-000030 June 2016[1]'
```

Figure 3.10: Printing d3 and its text attribute

Similar to **d1**, we see that it would be better to get only the span element's text.

21. Properly parse the date for this table entry by running the following code:

```
d3.find_all('span')
```

```
[1].text
```

```
>> '30 June 2016'
```

22. Now, we're ready to perform the full scrape by iterating over the row elements **<th>**. Run the following code:

```
data = []

for i, row in
    enumerate(body_content
        .find_all('tr')):
    ...

    ...

>> Ignoring row 101
    because len(data) != 3

>> Ignoring row 102
    because len(data) != 3
```

Note

For the complete code, refer to the following: <https://bit.ly/2EKMNbV>.

We iterate over the rows, ignoring any that contain more than three data elements. These rows will not correspond to data in the table we are interested in. Rows that do have

three data elements are assumed to be in the table, and we parse the text from these as identified during the testing.

The text parsing is done inside a **try/except** statement, which will catch any errors and allow this row to be skipped without stopping the iteration. Any rows that raise errors due to this statement should be looked at. The data for these could be recorded manually or accounted for by altering the scraping loop and re-running it. In this case, we'll ignore any errors for the sake of time.

23. Print the head of the scraped data list by running

```
print(data[:10]):
```

```
>> [['Albania', 1.25,  
      '4 May 2016'],  
      ['Angola', 16.0, '30  
      June 2016'],  
      ['Argentina', 26.25,  
      '11 April 2017'],  
      ['Armenia', 6.0, '14  
      February 2017'],
```

```
['Australia', 1.5, '2  
August 2016'],  
['Azerbaijan', 15.0,  
'9 September 2016'],  
['Bahamas', 4.0, '22  
December 2016'],  
['Bahrain', 1.5, '14  
June 2017'],  
['Bangladesh', 6.75,  
'14 January 2016'],  
['Belarus', 12.0, '28  
June 2017']]
```

24. We'll visualize this data later in the chapter. For now, save the data to a CSV file by running the following code:

```
f_path =  
'../data/countries/int  
erest-rates.csv' with  
open(f_path, 'w') as  
f:  
  
f.write('{};{};  
{ }\n'.format(*table_he  
aders)) for d in data:  
  
f.write('{};{};  
{ }\n'.format(*d))
```

Note that we are using semicolons to separate the fields.

ACTIVITY 3: WEB SCRAPING WITH JUPYTER NOTEBOOKS

You should have completed the previous exercise in this chapter.

In this activity, we are going to get the population of each country. Then, in the next topic, this will be visualized along with the interest rate data scraped in the previous exercise.

The page we look at in this activity is available here:
<http://www.worldometers.info/world-population/population-by-country/>.

Our aim is to apply the basic of web scrapping to a new web page and scrape some more data.

Note

*This page may have changed since this document was created.
If this URL no longer leads to a table of country populations,
please use this Wikipedia page instead:*
[https://en.wikipedia.org/wiki/List_of_countries_by_population\(United_Nations\)](https://en.wikipedia.org/wiki/List_of_countries_by_population(United_Nations)).

In order to do this, the following steps have to be executed:

1. Scrape the data from the web page.
2. In the **lesson-3-workbook.ipynb** Jupyter

Notebook, scroll to **Activity A: Web scraping with Python**.

3. Set the **url** variable and load an IFrame of our page in the notebook.
4. Close the IFrame by selecting the cell and clicking **Current Outputs** | Clear from the **Cell** menu in the Jupyter Notebook.
5. Request the page and load it as a **BeautifulSoup** object.
6. Print the H1 for the page.
7. Get and print the table headings.
8. Select first three columns and parse the text.
9. Get the data for a sample row.
10. How many columns of data do we have? Print the length of **row_data**.
11. Print the first elements of the row.
12. Select the data elements d1, d2, and d3.

13. Looking at the **row_data** output, we can find out how to correctly parse the data. Select the content of the **<a>** element in the first data element, and then simply get the text from the others.
14. Scrape and parse the table data.
15. Print the head of the scraped data.
16. Finally, save the data to a CSV file for later use.

Note

The detailed steps along with the solutions are presented in the Appendix A(pg. no. 160).

To summarize, we've seen how Jupyter Notebooks can be used for web scraping. We started this chapter by learning about HTTP methods and status codes. Then, we used the Requests library to actually perform HTTP requests with Python and saw how the BeautifulSoup library can be used to parse the HTML responses.

Our Jupyter Notebook turned out to be a great tool for this type of work. We were able to explore the results of our web requests and experiment with various HTML parsing techniques. We were also able to render the HTML and even load a live version of the web page inside the notebook!

In the next topic of this chapter, we shift to a completely new topic: interactive visualizations. We'll see how to create and

display interactive charts right inside the notebook, and use these charts as a way to explore the data we have just collected.

Interactive Visualizations

Visualizations are quite useful as a means of extracting information from a dataset. For example, with a bar graph it's very easy to distinguish the value distribution, compared to looking at the values in a table. Of course, as we have seen earlier in this book, they can be used to study patterns in the dataset that would otherwise be quite difficult to identify. Furthermore, they can be used to help explain a dataset to an unfamiliar party. If included in a blog post, for example, they can boost reader interest levels and be used to break up blocks of text.

When thinking about interactive visualizations, the benefits are similar to static visualizations, but enhanced because they allow for active exploration on the viewer's part. Not only do they allow the viewer to answer questions they may have about the data, they also think of new questions while exploring. This can benefit a separate party such as a blog reader or co-worker, but also a creator, as it allows for easy ad hoc exploration of the data in detail, without having to change any code.

In this topic, we'll discuss and show how to use Bokeh to build interactive visualizations in Jupyter. Prior to this, however, we'll briefly revisit pandas DataFrames, which play an important role in doing data visualization with Python.

BUILDING A DATAFRAME TO STORE AND ORGANIZE DATA

As we've seen time and time again in this book, pandas is an integral part of doing data science with Python and Jupyter Notebooks. DataFrames offer a way to organize and store labeled data, but more importantly, pandas provides time saving methods for transforming data within a DataFrame. Examples we have seen in this book include dropping duplicates, mapping dictionaries to columns, applying functions over columns, and filling in missing values.

With respect to visualizations, DataFrames offer methods for creating all sorts of matplotlib graphs, including **`df.plot.barh()`**, **`df.plot.hist()`**, and more. The interactive visualization library Bokeh previously relied on pandas DataFrames for their *high-level charts*. These worked similar to Seaborn, as we saw earlier in the previous chapter, where a DataFrame is passed to the plotting function along with the specific columns to plot. The most recent version of Bokeh, however, has dropped support for this behavior. Instead, plots are now created in much the same way as matplotlib, where the data can be stored in simple lists or NumPy arrays. The point of this discussion is that DataFrames are not entirely necessary, but still very helpful for organizing and manipulating the data prior to visualization.

EXERCISE 16: BUILDING AND MERGING PANDAS DATAFRAMES

Let's dive right into an exercise, where we'll continue working on the country data we scraped earlier. Recall that we extracted the central bank interest rates and populations of each country, and saved the results in CSV files. We'll load the data from these files and merge them into a DataFrame, which will then be used as the data source for the interactive visualizations to follow.

1. In the **lesson-3-**

workbook.ipynb of the Jupyter Notebook, scroll to the **Subtopic A: Building a DataFrame to store and organize data** subsection in the **Topic B** section.

We are first going to load the data from the CSV files, so that it's back to the state it was in after scraping. This will allow us to practice building DataFrames from Python objects, as opposed to using the **pd.read_csv** function.

Note

*When using **pd.read_csv**, the datatype for each column will be inferred from the string input. On the other hand, when using **pd.DataFrame** as we do here, the datatype is instead taken as the type of the input variables. In our case, as will be seen, we read the file and do not bother converting the variables to numeric or date-time until after instantiating the DataFrame.*

2. Load the CSV files into lists by

running the following code:

```
with
open('../data/countries/interest-rates.csv',
'r') as f:

int_rates_col_names =
next(f).split(',')

int_rates =
[line.split(',') for
line in f.read().
splitlines()]

with
open('../data/countries/populations.csv',
'r') as f:
populations_col_names
= next(f).split(',')

populations =
[line.split(',') for
line in f.read().
splitlines()]
```

3. Check what the resulting lists look like by running the next two cells. We should see an output similar to the following:

```
print(int_rates_col_names) int_rates[:5]
```

```
>> ['Country or
currency union',
'Central bank interest
...
...
['Indonesia', '263',
'991', '379', '1.10
%'],
['Brazil', '209',
'288', '278', '0.79
%']]
```

Now, the data is in a standard Python list structure, just as it was after scraping from the web pages in the previous sections. We're now going to create two DataFrames and merge them, so that all of the data is organized within one object.

4. Use the standard DataFrame constructor to create the two DataFrames by running the following code:

```
df_int_rates =
pd.DataFrame(int_rates
, columns=int_rates_
col_names)

df_populations =
```

```
pd.DataFrame(populations,
              columns=populations_column_names)
```

This isn't the first time we've used this function in this book. Here, we pass the lists of data (as seen previously) and the corresponding column names. The input data can also be of dictionary type, which can be useful when each column is contained in a separate list.

Next, we're going to clean up each DataFrame. Starting with the interest rates one, let's print the head and tail, and list the data types.

5. When displaying the entire DataFrame, the default maximum number of rows is 60 (for version 0.18.1). Let's reduce this to 10 by running the following code:

```
pd.options.display.max_rows = 10
```

6. Display the head and tail of the interest rates DataFrame by running the following code:

df_int_rates

	Country or currency union	Central bank interest rate (%)	Date of last change
0	Albania	1.25	4 May 2016
1	Angola	16.0	30 June 2016
2	Argentina	26.25	11 April 2017
3	Armenia	6.0	14 February 2017
4	Australia	1.5	2 August 2016
...	...	...	...
64	United States	1.25	14 June 2017
65	Uzbekistan	9.0	1 January 2015
66	Vietnam	6.25	7 July 2017
87	West African States	3.5	16 September 2013
88	Zambia	12.5	17 May 2017

89 rows x 3 columns

Figure 3.11: Table for interest rates by country

7. Print the data types by running:

```
df_int_rates.dtypes  
  
>> Country or currency  
union object  
  
>> Central bank  
interest rate (%)  
object  
  
>> Date of last change  
object  
  
>> dtype: object
```

Pandas has assigned each column

as a string datatype, which makes sense because the input variables were all strings. We'll want to change these to string, float, and datetime, respectively.

8. Convert to the proper datatypes by running the following code:

```
df_int_rates['Central  
bank interest rate  
(%)'] = \  
df_int_rates['Central  
bank interest rate  
(%)']\  
  
.astype(float,  
copy=False)  
  
df_int_rates['Date of  
last change'] = \  
pd.to_datetime(df_int_  
rates['Date of last  
change'])
```

We use **astype** to cast the Interest Rate values as floats, setting **copy=False** to save memory. Since the date values are given in such an easy-to-read format, these can be converted simply by using **pd.to_datetime**.

9. Check the new datatypes of each column by running the following code:

```
df_int_rates.dtypes  
  
>> Country or currency  
union object  
  
>> Central bank  
interest rate (%)  
float64  
  
>> Date of last change  
datetime64[ns]  
  
>> dtype: object
```

As can be seen, everything is now in the proper format.

10. Let's apply the same procedure to the other DataFrame. Run the next few cells to repeat the preceding steps for **df_populations**:

```
df_population
```

	Country (or dependency)	Population (2017)	Yearly Change
0	China	1,408,517,397	0.43 %
1	India	1,339,180,127	1.13 %
2	U.S.	324,459,463	0.71 %
3	Indonesia	263,991,379	1.10 %
4	Brazil	209,288,276	0.79 %
...	...	...	...
228	Saint Helena	4,049	0.35 %
229	Falkland Islands	2,910	0.00 %
230	Niue	1,618	-0.37 %
231	Tokelau	1,300	1.40 %
232	Holy See	792	-1.12 %

Figure 3.12: Table for population by country

Then, run this code:

```
df_populations['Population (2017)'] =
df_populations['Population (2017)']\
.str.replace(',', '', '')\
.astype(float,
copy=False)

df_populations['Yearly Change'] =
df_populations['Yearly Change']\
.str.rstrip('%')\
.astype(float,
```

```
copy=False)
```

To cast the numeric columns as a float, we had to first apply some modifications to the strings in this case. We stripped away any commas from the populations and removed the percent sign from the Yearly Change column, using string methods.

Now, we're going to merge the DataFrames on the country name for each row. Keep in mind that these are still the raw country names as scraped from the web, so there might be some work involved with matching the strings.

11. Merge the DataFrames by running the following code:

```
df_merge =  
pd.merge(df_population  
s,  
df_int_rates,  
left_on='Country (or  
dependency)',  
right_on='Country or  
currency union',  
how='outer')
```

df_merge

We pass the population data in the left DataFrame and the interest rates in the right one, performing an outer match on the country columns. This will result in **NaN** values where the two do not overlap.

12. For the sake of time, let's just look at the most populated countries to see whether we missed matching any. Ideally, we would want to check everything. Look at the most populous countries by running the following code:

```
df_merge.sort_values('
Population (2017)',
ascending=False)\
.head(10)
```

	Country (or dependency)	Population (2017)	Yearly Change	Country or currency union	Central bank interest rate (%)	Date of last change
0	China	1.408517e+09	0.43	China	1.75	2015-10-23
1	India	1.339180e+09	1.13	India	6.00	2017-08-02
2	U.S.	3.244535e+08	0.71	NaN	NaN	NaN
3	Indonesia	2.639914e+08	1.10	Indonesia	4.75	2016-10-20
4	Brazil	2.182883e+08	0.79	Brazil	7.25	2017-07-26
5	Pakistan	1.970160e+08	1.97	Pakistan	5.75	2016-05-21
6	Nigeria	1.908863e+08	2.63	Nigeria	14.00	2016-07-26
7	Bangladesh	1.646890e+08	1.05	Bangladesh	6.75	2016-01-14
8	Russia	1.439930e+08	0.02	Russia	9.00	2017-06-16
9	Mexico	1.291833e+08	1.27	Mexico	7.00	2017-06-22

Figure 3.13: The table for most populous countries

It looks like U.S. didn't match up.
This is because it's listed as *United States* in the interest rates data.

Let's remedy this.

13. Fix the label for U.S. in the populations table by running the following code:

```
col = 'Country (or  
dependency)'  
  
df_populations.loc[df_  
populations[col] ==  
'U.S.'] = 'United  
States'
```

We rename the country for the populations DataFrame with the use of the **loc** method to locate that row.

Now, let's merge the DataFrames properly.

14. Re-merge the DataFrames on the country names, but this time use an inner merge to remove the **NaN** values:

```
df_merge =  
pd.merge(df_population  
s,  
df_int_rates,  
left_on='Country (or
```

```
dependency)',  
right_on='Country or  
currency union',  
  
how='inner')
```

15. We are left with two identical columns in the merged DataFrame. Drop one of them by running the following code:

```
del df_merge['Country  
or currency union']
```

16. Rename the columns by running the following code:

```
name_map = {'Country  
(or dependency)':  
'Country', 'Population  
(2017)': 'Population',  
'Central bank interest  
rate (%)': 'Interest  
rate'}  
  
df_merge =  
df_merge.rename(column  
s=name_map)
```

We are left with the following merged and cleaned DataFrame:

	Country	Population	Yearly Change	Interest rate	Date of last change
0	China	1.408517e+09	0.43	1.75	2015-10-23
1	India	1.339180e+09	1.13	6.00	2017-08-02
2	United States	3.244595e+08	0.71	1.25	2017-06-14
3	Indonesia	2.639914e+08	1.10	4.75	2016-10-20
4	Brazil	2.092883e+08	0.79	7.25	2017-07-26
...	...	...	...	...	...
76	Mauritius	1.265138e+06	0.24	4.00	2016-07-20
77	Fiji	9.055020e+05	0.75	0.50	2011-11-02
78	Bahamas	3.953610e+05	1.06	4.00	2016-12-22
79	Iceland	3.350250e+05	0.77	4.50	2017-06-14
80	Samoa	1.964400e+05	0.67	0.14	2016-07-01

81 rows x 5 columns

Figure 3.14: Ouput after cleaning and merging tables

- Now that we have all the data in a nicely organized table, we can move on to the fun part: visualizing it. Let's save this table to a CSV file for later use, and then move on to discuss how visualizations can be created with Bokeh.

Write the merged data to a CSV file for later use with the following code:

```
df_merge.to_csv(' ../da
```

```
ta/countries/merged.csv', index=False)
```

INTRODUCTION TO BOKEH

Bokeh is an interactive visualization library for Python. Its goal is to provide similar functionality to D3, the popular interactive visualization library for JavaScript. Bokeh functions very differently than D3, which is not surprising given the differences between Python and JavaScript. Overall, it's much simpler and it doesn't allow nearly as much customization as D3 does. This works to its advantage though, as it's much easier to use, and it still boasts an excellent suite of features that we'll explore in this section.

Let's dive right into a quick exercise with the Jupyter Notebook and introduce Bokeh by example.

Note

*There is good documentation online for Bokeh, but much of it is outdated. Searching something like Bokeh bar plot in Google still tends to turn up documentation for legacy modules that no longer exist, for example, the high-level plotting tools that used to be available through **bokeh.charts** (prior to version 0.12.0). These are the ones that take pandas DataFrames as input in much the same way that Seaborn plotting functions do. Removing the high-level plotting tools module has simplified Bokeh, and will allow for more focused development going forward. Now, the plotting tools are largely grouped into the **bokeh.plotting** module, as will be seen in the next exercise and following activity.*

EXERCISE 17:

INTRODUCTION TO INTERACTIVE VISUALIZATION WITH BOKEH

We'll load the required Bokeh modules and show some simple interactive plots that can be made with Bokeh. Please note that the examples in this book have been designed using version 0.12.10 of Bokeh.

1. In the **lesson-3-workbook.ipynb** Jupyter notebook, scroll to **Subtopic B: Introduction to Bokeh.**
2. Like scikit-learn, Bokeh modules are usually loaded in pieces (unlike pandas, for example, where the whole library is loaded at once). Import some basic plotting modules by running the following code:

```
from bokeh.plotting
import figure, show,
output_notebook
output_notebook()
```

We need to run **output_notebook()** in order to render the interactive visuals within the Jupyter notebook.

3. Generate random data to plot by running the following code:

```
np.random.seed(30)

data =
pd.Series(np.random.randn(200),
index=list(range(200))\
.cumsum() x =
data.index
y = data.values
```

The random data is generated using the cumulative sum of a random set of numbers that are distributed about zero. The effect is a trend that looks similar to a stock price time series, for example.

4. Plot the data with a line plot in Bokeh by running the following code:

```
p =
figure(title='Example
plot',
x_axis_label='x',
y_axis_
label='y')p.line(x, y,
legend='Random trend')
```

show(p)

Example plot

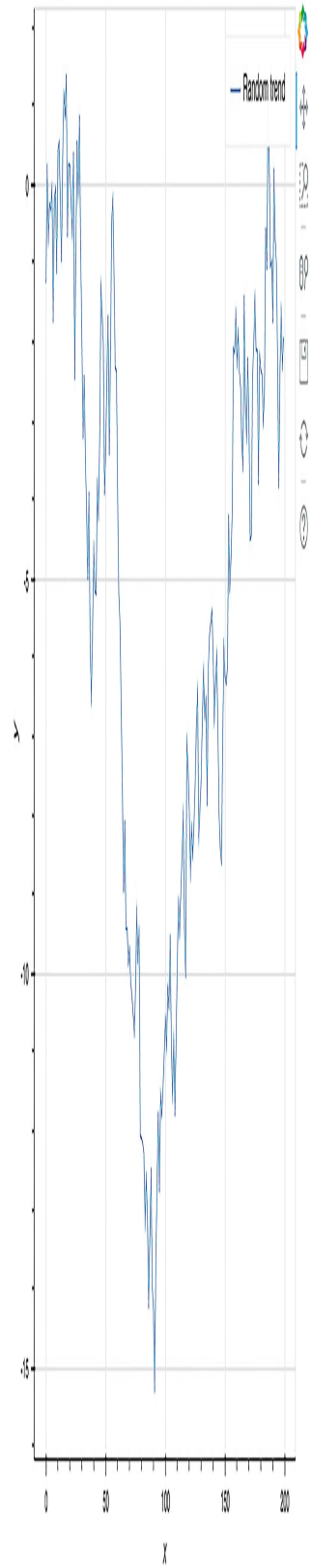


Figure 3.15: An example data plot

We instantiate the figure, as referenced by the variable **p**, and then plot a line. Running this in Jupyter yields an interactive figure with various options along the right-hand side.

The top three options (as of version 0.12.10) are **Pan**, **Box Zoom**, and **Wheel Zoom**. Play around with these and experiment with how they work. Use the reset option to reload the default plot limits.

5. Other plots can be created with the alternative methods of **figure**. Draw a scatter plot by running the following code, where we replace **line** in the preceding code with **circle**:

```
size =  
np.random.rand(200) *  
5  
  
p =  
figure(title='Example  
plot',  
x_axis_label='x',  
y_axis_label='y')
```

```
p.circle(x, y,  
radius=size,  
alpha=0.5,  
legend='Random dots')  
  
show(p)
```

Example plot

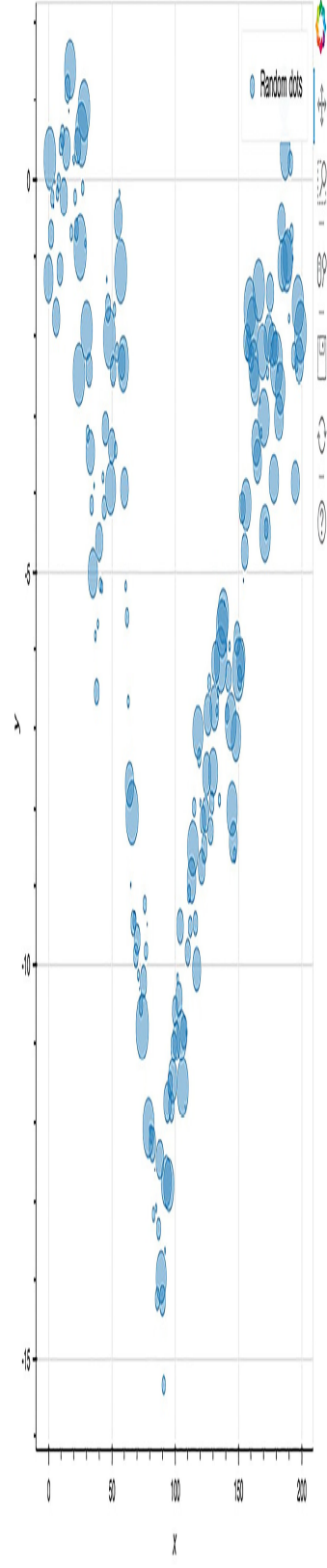


Figure 3.16: An example scatter plot

Here, we've specified the size of each circle using a random set of numbers. A very enticing feature of interactive visualizations is the tooltip. This is a hover tool that allows the user to get information about a point by hovering over it.

6. In order to add this tool, we're going to use a slightly different method for creating the plot. This will require us to import a couple of new libraries. Run the following code:

```
from bokeh.plotting
import
ColumnDataSource from
bokeh.models import
HoverTool
```

This time, we'll create a data source to pass to the plotting method. This can contain metadata, which can be included in the visualization via the hover tool.

7. Create random labels and plot the interactive visualization with a hover tool by running the following

code:

```
source =  
ColumnDataSource(data=  
dict( x=x,  
y=y,  
...  
...  
source=source,  
legend='Random dots')  
show(p)
```

Note

For the complete code, refer to the following: <https://bit.ly/2RhpU1r>.

Example plot

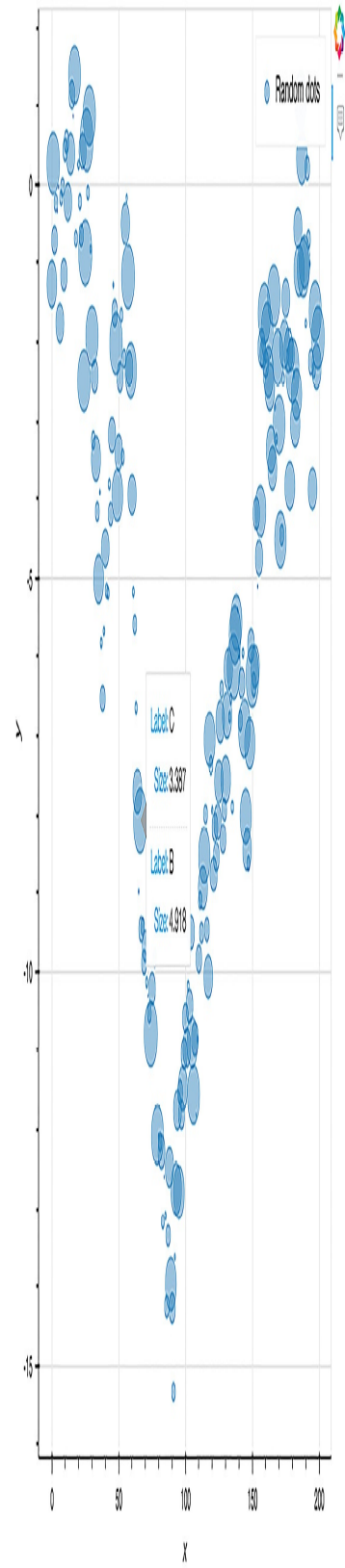


Figure 3.17: A random scatter plot with labels

We define a data source for the plot by passing a dictionary of key/value pairs to the **ColumnDataSource** constructor. This source includes the *x* location, *y* location, and size of each point, along with the random letter **A**, **B**, or **C** for each point. These random letters are assigned as labels for the hover tool, which will also display the size of each point. The **Hover Tool** is then added to the figure, and the data is retrieved from each element through the specific plotting method, which is `circle` in this case. The result is that we are now able to hover over the points and see the data we've selected for the **Hover Tool**! We notice, by looking at the toolbar to the right of the plot, that by explicitly including the **Hover Tool**, the others have disappeared. These can be included by manually adding them to the list of tool objects that gets passed to **`bokeh.plotting.figure`**.

8. Add pan, zoom, and reset tools to

the plot by running the following code:

```
from bokeh.models
import PanTool,
BoxZoomTool,
WheelZoomTool,
ResetTool

...

...

legend='Random dots')

show(p)
```

This code is identical to what was previously shown except for the **tools** variable, which now references several new tools we've imported from the Bokeh library.

We'll stop the introductory exercise here, but we'll continue creating and exploring plots in the following activity.

ACTIVITY 4: EXPLORING DATA WITH INTERACTIVE VISUALIZATIONS

You should have completed the previous exercise in order to complete this activity.

We'll pick up using Bokeh right where we left off with the

previous exercise, except instead of using the randomly generated data seen there, we'll instead use the data we scraped from the web in the first part of this chapter. Our aim is to use Bokeh to create interactive visualizations of our scraped data.

In order to do so, we need to execute the following steps:

1. In the **lesson-3-workbook.ipynb** file, scroll to the **Activity B: Interactive visualizations with Bokeh** section.
2. Load the previously scraped, merged, and cleaned web page data
3. Recall what the data looks like by displaying the DataFrame.
4. Draw a scatter plot of the population as a function of the interest rate.
5. In the data, we see some clear outliers with high populations. Hover over these to see what they are. Select the Box Zoom tool and alter the viewing window to better see the majority of the data.
6. Some of the lower population countries appear to have negative

interest rates. Select the **Wheel Zoom** tool and use it to zoom in on this region. Use the **Pan** tool to re-center the plot, if needed, so that the negative interest rate samples are in view. Hover over some of these and see what countries they correspond to.

7. Add a **Year of last change** column to the DataFrame and add a color based on the date of last interest rate change
8. Create a map to group the last change date into color categories.
9. Create the colored visualization.
10. Looking for patterns, zoom in on the lower population countries.
11. Plot the interest rate as a function of the year-over-year population change by running the following code.
12. Determine the line of best fit for the previously plotted relationship.
13. Re-plot the output obtained in the preceding step and add a line of

best fit.

14. Explore the plot by using the zoom tools and hovering over interesting samples.

Note

The detailed steps along with the solutions are presented in the Appendix A(pg. no. 163).

Summary

In this chapter, we scraped web page tables and then used interactive visualizations to study the data.

We started by looking at how HTTP requests work, focusing on GET requests and their response status codes. Then, we went into the Jupyter Notebook and made HTTP requests with Python using the Requests library. We saw how Jupyter can be used to render HTML in the notebook, along with actual web pages that can be interacted with. After making requests, we saw how BeautifulSoup can be used to parse text from the HTML, and used this library to scrape tabular data.

After scraping two tables of data, we stored them in pandas DataFrames. The first table contained the central bank interest rates for each country and the second table contained the populations. We combined these into a single table that was then used to create interactive visualizations.

Finally, we used Bokeh to render interactive visualizations in Jupyter. We saw how to use the Bokeh API to create various customized plots and made scatter plots with specific interactive abilities such as zoom, pan, and hover. In terms of

customization, we explicitly showed how to set the point radius and color for each data sample.

Furthermore, when using Bokeh to explore the scraped population data, the tooltip was utilized to show country names and associated data when hovering over the points.

Congratulations for completing this introductory course on data science using Jupyter Notebooks! Regardless of your experience with Jupyter and Python coming into the book, you've learned some useful and applicable skills for practical data science!

Before finishing up, let's quickly recap the topics we've covered in this book.

The first chapter was an introduction to the Jupyter Notebook platform, where we covered all of the fundamentals. We learned about the interface and how to use and install magic functions. Then, we introduced the Python libraries we'll be using and walked through an exploratory analysis of the *Boston housing* dataset.

In the second chapter, we focused on doing machine learning with Jupyter. We first discussed the steps for developing a predictive analytics plan, and then looked at a few different types of models including SVM, a KNN classifier, and Random Forests.

Working with an *employee retention* dataset, we applied data cleaning methods and then trained models to predict whether an employee has left or not. We also explored more advanced topics such as overfitting, k-fold cross-validation, and validation curves.

Finally, in the third chapter, we shifted briefly from data analysis to data collection using web scraping and saw how to make HTTP requests and parse the HTML responses in Jupyter. Then, we finished up the book by using interactive visualizations to explore our collected data.

Appendix A

About

This section is included to assist the students to perform the activities present in the book. It includes detailed steps that are to be performed by the students to complete and achieve the objectives of the activity.

Chapter 1: Jupyter Fundamentals

ACTIVITY 1: BUILDING A THIRD-ORDER POLYNOMIAL MODEL

1. Scroll to the empty cells at the bottom of **Subtopic C** in your Jupyter Notebook.
2. These will be found beneath the linear-model **MSE** calculation cell under the **Activity** heading.

Note

You should fill these empty cells in with code as we complete the activity. You may need to insert new cells as these become filled up; please do so as needed.

3. We will first pull out our dependent feature from and target variable from **df** . using ...