Jason S. Schwarz · Chris Chapman · Elea McDonnell Feit

Python for Marketing Research and Analytics

Springer

Python for Marketing Research and Analytics

Jason S. Schwarz • Chris Chapman • Elea McDonnell Feit

Python for Marketing Research and Analytics

Springer

Jason S. Schwarz
Google
Nashville, TN, USA

Chris Chapman
Google
Seattle, WA, USA

Elea McDonnell Feit
Drexel University
Philadelphia, PA, USA

ISBN 978-3-030-49719-4 ISBN 978-3-030-49720-0 (eBook)
https://doi.org/10.1007/978-3-030-49720-0

© Springer Nature Switzerland AG 2020
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

We are here to help you learn Python for marketing research and analytics.

Python is a great choice for marketing analysts. It offers advanced capabilities for fitting statistical models. It is extensible and is able to process data from many different systems, in a variety of forms, for both small and large datasets. The Python ecosystem includes a vast range of established and emerging statistical methods as well as visualization techniques. Yet its use in marketing lags other fields such as econometrics, bioinformatics, and computer science. With your help, we hope to change that!

This book is designed for two audiences: practicing marketing researchers and analysts who want to learn Python and students or researchers from other fields who want to review selected marketing topics in a Python context.

What are the prerequisites? Simply that you are interested in Python for marketing, are conceptually familiar with basic statistical models such as linear regression, and are willing to engage in hands-on learning. This book will be particularly helpful to analysts who have some degree of programming experience and wish to learn Python. In Chap. 1, we describe additional reasons to use Python (and a few reasons perhaps *not* to use Python).

The *hands-on* part is important. We teach concepts gradually in a sequence across the first seven chapters and ask you to *type* our examples as you work; this book is *not* a cookbook style reference. We spend some time (as little as possible) in Part I on the basics of the Python language and then turn in Part II to applied, real-world marketing analytics problems. Part III presents a few advanced marketing topics. Every chapter shows off the power of Python, and we hope each one will teach you something new and interesting.

Specific features of this book are:

- It is organized around marketing research tasks. Instead of generic examples, we put methods into the context of marketing questions.
- We presume only basic statistics knowledge and use a minimum of mathematics. This book is designed to be approachable for practitioners and does not dwell on equations or mathematical details of statistical models (although we give references to those texts).
- This is a didactic book that explains statistical concepts and the Python code. We want you to understand what we are doing and learn how to avoid common problems in both statistics and Python. We intend the book to be *readable* and to fulfill a different need than references and cookbooks available elsewhere.
- The applied chapters demonstrate progressive model building. We do not present "the answer" but instead show how an analyst might realistically conduct analyses in successive steps where multiple models are compared for statistical strength and practical utility.
- The chapters include visualization as a part of core analyses. We do not regard visualization as a standalone topic; rather, we believe it is an integral part of data exploration and model building.
- Most of the analyses use simulated data, which provides practice in the Python language along with additional insight into the structure of marketing data. If you are inclined, you can change the data simulation and see how the statistical models are affected.
- Where appropriate, we call out more advanced material on programming or models so that you may either skip it or read it, as you find appropriate. These sections are indicated by * in their titles (such as *This is an advanced section**).

What do we *not* cover? For one, this book teaches *Python* for marketing and does not teach marketing research in itself. We discuss many marketing topics but omit others that would simply repeat the analytic methods in Python. As noted above, we approach statistical models from a conceptual point of view and skip the mathematics. A few specialized topics have been omitted due to complexity and space; these include customer lifetime value models and econometric time series models.

Overall, we believe the analyses here represent a great sample of marketing research and analytics practice. If you learn to perform these, you will be well equipped to apply Python in many areas of marketing.

For another, this book teaches Python for *marketing* and not all of the complexity or subtlety of the Python programming language or of programming generally. We present the basics of Python programming that are needed to successfully analyze marketing data, but there is much more to Python than we present here.

Companion Book: *R for Marketing Research and Analytics*

This book is closely related to *R for Marketing Research and Analytics* (Chapman and Feit 2019) and shares many datasets and sections of didactic explanation of methods with that R text. In some ways, this Python book and the R text are mutual "translations" of one another from R and Python, respectively.

This was a deliberate choice that we hope will make it easy for readers to move between Python and R. If you understand a method in one language, the companion book will demonstrate how to perform a similar or identical analysis in the other language. For example, if you learned to program R from that text, you will be able to learn Python rapidly using this book. And if you master analyses in Python here, you will be able easily to perform most of the same analyses in R using that text. You will already be familiar with most of the theoretical sections and datasets and can focus on the language.

At the same time, each book has a few topics that are unique to its language and not covered in the other. This Python text has somewhat more emphasis on programming and writing custom functions. The R text includes methods for choice-based conjoint analysis (discrete choice models), market basket analysis with association rules, and methods to model behavior sequences such as weblogs. Those differences reflect the general situation in Python to have somewhat fewer yet often more stable and higher performance tools, contrasting an emphasis in R on a vast ecosystem of tools. In short, there are great reasons to learn both Python and R! The paired texts have been designed to make that easier.

Acknowledgements

We want to give special thanks here to people who made this book possible. First are all the students from our tutorials and classes over the years. They provided valuable feedback, and we hope their experiences will benefit you.

Jason's and Chris's colleagues in the research community at Google provided extensive feedback on portions of the book. We thank the following Googlers: Javier Bargas, Mario Callegaro, Xu Gao, Rohan Gifford, Michael Gilbert, Xiaoyu He, Tim Hesterberg, Shankar Kumar, Kishan Panchal, Katrina Panovich, Michael Quinn, David Remus, Marta Rey-Babarro, Dan Russell, Rory Sayres, Angela Schörgendorfer, Micha Segeritz, Bob Silverstein, Matt Small, Gill Ward, John Webb, Rui Zhong, and Yori Zwols. Their encouragement and reviews have greatly improved the book.

In the broader community, we had valuable feedback from Lynd Bacon, Marianna Dizik, Dennis Fok, Norman Lemke, Paul Litvak, Kerry Rodden, Steven Scott, and Randy Zwitch.

The staff and editors at Springer helped us smooth the process, especially Senior Editor Lorraine Klimowich.

Much of this book was written in public and university libraries, and we thank them for their hospitality alongside their unsurpassed literary resources. Portions of the book were written during pleasant days at the New Orleans Public Library, New York Public Library, Christoph Keller, Jr. Library at the General Theological Seminary in New York, University of California San Diego Giesel Library, University of Washington Suzzallo and Allen Libraries, Sunnyvale Public Library, and the Tokyo Metropolitan Central Library.

Most importantly, we thank *you*, the reader. We are glad you have decided to investigate Python, and we hope to repay your effort. Let us start!

Nashville, TN, USA — Jason S. Schwarz
Seattle, WA, USA — Chris Chapman
Philadelphia, PA, USA — Elea McDonnell Feit
August 2020

Contents

Part I Basics of Python

1 Welcome to Python ... 3
- 1.1 What is Python? ... 3
- 1.2 Why Python? ... 3
  - 1.2.1 Python vs. R, Julia, and Others ... 4
- 1.3 Why Not Python? ... 4
- 1.4 When to Use Python? ... 5
- 1.5 Using This Book ... 5
  - 1.5.1 About the Text ... 5
  - 1.5.2 About the Data ... 6
  - 1.5.3 Online Material ... 6
  - 1.5.4 When Things Go Wrong ... 7
- 1.6 Key Points ... 7

2 An Overview of Python ... 9
- 2.1 Getting Started ... 9
  - 2.1.1 Notebooks ... 9
  - 2.1.2 Installing Python Locally ... 10
  - 2.1.3 Running Python Locally ... 10
- 2.2 A Quick Tour of Python Data Analysis Capabilities ... 11
- 2.3 Basics of Working with Python Commands ... 14
  - 2.3.1 Python Style ... 15
- 2.4 Basic Types ... 15
  - 2.4.1 Objects and Type ... 15
  - 2.4.2 Booleans ... 16
  - 2.4.3 Numeric Types ... 16
  - 2.4.4 Sequence Types ... 17
  - 2.4.5 Text Type: String ... 19
  - 2.4.6 Set Type ... 20
  - 2.4.7 Mapping Type ... 21
  - 2.4.8 Functions, Classes, and Methods ... 22
  - 2.4.9 Modules and Packages ... 25
  - 2.4.10 Control Flow Statements ... 26
  - 2.4.11 Help! A Brief Detour ... 31
- 2.5 Data Science Packages ... 33
  - 2.5.1 NumPy ... 33
  - 2.5.2 Using Python for Mathematical Computation ... 35
  - 2.5.3 pandas ... 36
  - 2.5.4 Missing Values ... 40

2.6 Loading and Saving Data ... 41
2.6.1 Saving Python Objects: Pickle ... 42
2.6.2 Importing and Exporting Data ... 43
2.6.3 Using Colab: Importing and Exporting Data ... 44
2.7 Clean Up! ... 44
2.8 Learning More* ... 45
2.9 Key Points ... 45

Part II Fundamentals of Data Analysis

3 Describing Data ... 49
3.1 Simulating Data ... 49
3.1.1 Store Data: Setting the Structure ... 49
3.1.2 Store Data: Simulating Data Points ... 53
3.2 Functions to Summarize a Variable ... 56
3.2.1 Language Brief: `groupby()` ... 56
3.2.2 Discrete Variables ... 57
3.2.3 Continuous Variables ... 59
3.3 Summarizing Dataframes ... 61
3.3.1 `describe()` ... 61
3.3.2 Recommended Approach to Inspecting Data ... 62
3.3.3 `apply()`* ... 63
3.4 Single Variable Visualization ... 65
3.4.1 Histograms ... 65
3.4.2 Boxplots ... 69
3.4.3 QQ Plot to Check Normality* ... 70
3.4.4 Cumulative Distribution* ... 71
3.4.5 Maps ... 74
3.5 Learning More* ... 75
3.6 Key Points ... 75

4 Relationships Between Continuous Variables ... 77
4.1 Retailer Data ... 77
4.1.1 Simulating the Data ... 77
4.1.2 Simulating Online and In-store Sales Data ... 79
4.1.3 Simulating Satisfaction Survey Responses ... 80
4.1.4 Simulating Non-response Data ... 81
4.2 Exploring Associations between Variables with Scatterplots ... 82
4.2.1 Creating a Basic Scatterplot with `plot()` ... 83
4.2.2 Color-Coding Points on a Scatterplot ... 86
4.2.3 Plotting on a Log Scale ... 87
4.3 Combining Plots in a Single Graphics Object ... 88
4.4 Scatterplot Matrices ... 90
4.4.1 `scatter_matrix()` ... 90
4.4.2 `PairGrid()` ... 90
4.5 Correlation Coefficients ... 93
4.5.1 Correlation Tests ... 94
4.5.2 Correlation Matrices ... 94
4.5.3 Transforming Variables before Computing Correlations ... 95
4.5.4 Typical Marketing Data Transformations ... 98
4.5.5 Box-Cox Transformations* ... 98
4.6 Exploring Associations in Survey Responses* ... 100
4.6.1 Jitter: Make Ordinal Plots More Informative* ... 101
4.7 Learning More* ... 101
4.8 Key Points ... 101

5 Comparing Groups: Tables and Visualizations 103
5.1 Simulating Consumer Segment Data 103
5.1.1 Segment Data Definition 104
5.1.2 Final Segment Data Generation 106
5.2 Finding Descriptives by Group 109
5.2.1 Descriptives for Two-way Groups 112
5.2.2 Visualization by Group: Frequencies and Proportions 114
5.2.3 Visualization by Group: Continuous Data 116
5.2.4 Bringing It All Together 119
5.3 Learning More* 119
5.4 Key Points 119

6 Comparing Groups: Statistical Tests 121
6.1 Data for Comparing Groups 121
6.2 Testing Group Frequencies: `scipy.stats.chisquare()` 122
6.3 Testing Observed Proportions: `binom_test()` 125
6.3.1 About Confidence Intervals 126
6.3.2 More About `binom_test()` and Binomial Distributions 126
6.4 Testing Group Means: t Test 127
6.5 Testing Multiple Group Means: Analysis of Variance (ANOVA) 130
6.5.1 A Brief Introduction to Formula Syntax 130
6.5.2 ANOVA 130
6.5.3 Model Comparison in ANOVA* 132
6.5.4 Visualizing Group Confidence Intervals 132
6.6 Learning More* 135
6.7 Key Points 136

7 Identifying Drivers of Outcomes: Linear Models 137
7.1 Amusement Park Data 137
7.1.1 Simulating the Amusement Park Data 138
7.2 Fitting Linear Models with `ols()` 140
7.2.1 Preliminary Data Inspection 140
7.2.2 Recap: Bivariate Association 141
7.2.3 Linear Model with a Single Predictor 143
7.2.4 `ols` Objects 145
7.2.5 Checking Model Fit 147
7.3 Fitting Linear Models with Multiple Predictors 152
7.3.1 Comparing Models 153
7.3.2 Using a Model to Make Predictions 155
7.3.3 Standardizing the Predictors 156
7.4 Using Factors as Predictors 157
7.5 Interaction Terms 160
7.5.1 Language Brief: Advanced Formula Syntax* 162
7.5.2 Caution! Overfitting 162
7.5.3 Recommended Procedure for Linear Model Fitting 163
7.6 Learning More* 164
7.7 Key Points 164

8 Additional Linear Modeling Topics 167
8.1 Handling Highly Correlated Variables 167
8.1.1 An Initial Linear Model of Online Spend 167
8.1.2 Remediating Collinearity 170
8.2 Linear Models for Binary Outcomes: Logistic Regression 173
8.2.1 Basics of the Logistic Regression Model 173
8.2.2 Data for Logistic Regression of Season Passes 174
8.2.3 Sales Table Data 175

8.2.4 Fitting a Logistic Regression Model 177
8.2.5 Reconsidering the Model 178
8.2.6 Additional Discussion 181
8.3 An Introduction to Hierarchical Models 182
8.3.1 Some HLM Concepts 182
8.3.2 Ratings-Based Conjoint Analysis for the Amusement Park 182
8.3.3 Simulating Ratings-Based Conjoint Data 183
8.3.4 An Initial Linear Model 185
8.3.5 Hierarchical Linear Model with `mixedlm` 186
8.3.6 The Complete Hierarchical Linear Model 188
8.3.7 Interpreting HLM 190
8.3.8 Conclusion for HLM 190
8.4 Learning More* 191
8.5 Key Points 192

Part III Advanced Data Analysis

9 Reducing Data Complexity 195
9.1 Consumer Brand Rating Data 195
9.1.1 Rescaling the Data 197
9.1.2 Correlation Between Attributes 198
9.1.3 Aggregate Mean Ratings by Brand 198
9.2 Principal Component Analysis and Perceptual Maps 200
9.2.1 PCA Example 201
9.2.2 Visualizing PCA 203
9.2.3 PCA for Brand Ratings 206
9.2.4 Perceptual Map of the Brands 207
9.2.5 Cautions with Perceptual Maps 210
9.3 Exploratory Factor Analysis 210
9.3.1 Basic EFA Concepts 211
9.3.2 Finding an EFA Solution 211
9.3.3 EFA Rotations 213
9.3.4 Using Factor Scores for Brands 214
9.4 Multidimensional Scaling 216
9.4.1 Non-metric MDS 217
9.4.2 Visualization Using Low-Dimensional Embeddings 219
9.5 Learning More* 221
9.6 Key Points 221

10 Segmentation: Unsupervised Clustering Methods for Exploring Subpopulations 223
10.1 Segmentation Philosophy 223
10.1.1 The Difficulty of Segmentation 223
10.1.2 Segmentation as Clustering and Classification 224
10.2 Segmentation Data 224
10.3 Clustering 225
10.3.1 The Steps of Clustering 226
10.3.2 Hierarchical Clustering 228
10.3.3 Hierarchical Clustering Continued: Groups from `fcluster` 232
10.3.4 Mean-Based Clustering: `k_means()` 235
10.3.5 Model-Based Clustering: `GaussianMixture()` 238
10.3.6 Recap of Clustering 240
10.4 Learning More* 241
10.5 Key Points 241

11 Classification: Assigning Observations to Known Categories 243
11.1 Classification 243
11.1.1 Naive Bayes Classification: `GaussianNB()` 243
11.1.2 Random Forest Classification: `RandomForestClassifier()` 250
11.1.3 Random Forest Variable Importance 256
11.2 Prediction: Identifying Potential Customers* 256
11.3 Learning More* 260
11.4 Key Points 261

12 Conclusion 263

References 265

Index 267

Part I
Basics of Python

Chapter 1
Welcome to Python

1.1 What is Python?

Python is a general-purpose programming language. It has increasingly become the language of choice not only for teaching programming, given its simple syntax and great readability, but for programming applications of all kinds, ranging from data analysis and data science to full stack web development.

If you are a marketing analyst, you have no doubt heard of Python. You may have tried Python or another language like R and become frustrated and confused, after which you returned to other tools that are "good enough." You may know that Python uses a command line and dislike that. Or you may be convinced of Python's advantages for experts but worry that you don't have time to learn or use it.

Or if you come from a programming rather than market analyst background and have little experience with formal analysis, you might have tried to explore complex datasets but gotten frustrated by data transformations, statistics, or visualization.

We are here to help! Our goal is to present *just the essentials*, in the *minimal necessary time*, with *hands-on learning* so you will come up to speed as quickly as possible to be productive analyzing data in Python. In addition, we'll cover a few advanced topics that demonstrate the power of Python and might teach advanced users some new skills.

A key thing to realize is that *Python is a programming language*. It is *not* a "statistics program" like SPSS, SAS, JMP, or Minitab, and doesn't wish to be one. It is extremely flexible; in Python you can write code to fill nearly any requirement, from data ingestions and transformation to statistical analysis and visualization. Python enjoys a thriving open source community. Scientists and statisticians have added a huge amount of statistical and scientific computing functionality to Python through new libraries. These libraries add functionality seen in specialized languages like R or Matlab, turning Python into a powerful tool for data science.

1.2 Why Python?

Python was designed with a priority of *code readability*. Readability is about the ease of quickly understanding what code is doing when reading it. In Python, the functionality of code should be obvious. Why is that important? It's important because code can easily get complicated. Approaching coding with a goal of simplicity and straightforwardness makes for better, less buggy, and more shareable code.

This is the reason why Python is often the first language taught in schools. Programmers sometimes joke that Python is "just pseudocode," meaning that it looks almost exactly like what you would write while you were *designing* your code, not actually implementing it. There is no complicated syntax, no memory management, and it is not strictly typed (See Sect. 2.4.1). And systematic whitespace requirements ensure that code is formatted consistently.

Python balances this simplicity with flexibility, power, and speed. There's a reason that Python recently has been the fastest growing programming language in absolute terms (Robinson 2017). Python is useful not only for scripting and web frameworks, but also for data pipelines, machine learning, and data analysis.

A great thing about Python is that it integrates well into production environments. So if you want to automate a process, such as generating a report, scoring a data stream based on a model, or sending an email based on events, those tasks can

© Springer Nature Switzerland AG 2020

J. S. Schwarz et al., *Python for Marketing Research and Analytics*, https://doi.org/10.1007/978-3-030-49720-0_1

usually be prototyped in Python and then put directly into production in Python, streamlining the development process. (Although, this depends somewhat on the tech stack you use in production).

For analysts, Python offers a large and diverse set of analytic tools and statistical methods. It allows you to write analyses that can be reused and that extend the Python functionality itself. It runs on most operating systems and interfaces well with data systems such as online data and SQL databases. Python offers beautiful and powerful plotting functions that are able to produce graphics vastly more tailored and informative than typical spreadsheet charts. Putting all of those together, Python can vastly improve an analyst's overall productivity.

Then there is the community. Many Python users are enthusiasts who love to help others and are rewarded in turn by the simple joy of solving problems and the fact that they often learn something new. Python is a dynamic system created by its users, and there is always something new to learn. Knowledge of Python is a valuable skill in demand for analytics jobs at a growing number of top companies.

The code for functions you use in Python is also inspectable; you may choose to trust it, yet you are also free to verify. All of its core code and most packages that people contribute are open source. You can examine the code to see exactly how analyses work and what is happening under the hood.

Finally, Python is free. It is a labor of love and professional pride for the Python Core Developers. As with all masterpieces, the quality of their devotion is evident in the final work.

1.2.1 Python vs. R, Julia, and Others

If you are new to programming, you might wonder whether to learn Python or R . . . or Julia, Matlab, Ruby, Go, Java, C++, Fortran, or others. Each of those languages is a great choice, depending on a few differentiating factors.

If your work involves large data transformation, exploration, visualization, and statistical analysis, then Python is a great choice. If machine learning is relevant for you, several of the most powerful machine learning libraries are Python-native, such as Theano, Keras, PyTorch, and Tensorflow. If you want your analytic work to go into production and integrate with a larger system (such as a product or a web site), then, again, Python is a great choice.

Another factor is whether you wish to program more generally beyond analytics, such as writing apps. Python is an excellent general purpose language. It is more approachable than C++, while it also has broader support for statistics and analytics than Go, Java, or Ruby.

If you want to leverage advanced statistics, such as Bayesian analyses or structural equation modeling, then R is unmatched (Chapman and Feit 2019). If high performance is essential to you, such as working with massive datasets or models with high mathematical complexity, then Julia is an excellent choice (Lauwens and Downey 2019). Go is also designed for massive scalability.

If you often do a lot of directly mathematical work, such as writing equations for models, then Python is a fine choice, although Julia, R, Matlab, Mathematica, or even Fortran might be more comfortable for you.

Finally, there is the question of your environment. If you work with others who program, it will be advantageous to use a language they prefer, so you can get expert help. At the same time, most languages interact well with others. For example, it is quite easy to write analytic code in R and to access it from Python (and vice versa). C++ code can be embedded in Python, and in many other languages, when needed (Foundation 2020). In other words, if you learn Python, it will be usable elsewhere. Many programmers end up using several languages and find that transitioning among them is not difficult.

In short, for analyses with high flexibility and a straightforward programming environment, Python is a great choice.

1.3 Why Not Python?

It's hard for us to imagine NOT using Python for analysis, but of course many people don't, so what are the reasons not to use it?

One reason not to use Python is this: until you've mastered the basics of the language, many simple analyses are cumbersome to do in Python. If you're new to Python and want a table of means, cross-tabs, or a t-test, it may be frustrating to figure out how to get them. Python is about power, flexibility, control, iterative analyses, and cutting-edge methods, not point-and-click deliverables.

Another reason is if you do not like programming. If you're new to programming, Python is a great place to start. But if you've tried programming before and didn't enjoy it, Python may be a challenge as well. Our job is to help you as much as

we can, and we will try hard to teach basic Python to you. However, not everyone enjoys programming. On the other hand, if you're an experienced coder Python will seem simple (perhaps deceptively so), and we will help you avoid a few pitfalls.

One other concern about Python is the unpredictability of its ecosystem. With packages contributed by thousands of developers, there are priceless contributions along with others that are mediocre or flawed, although that is rare with the major packages (e.g. NumPy, pandas, scikit-learn, statsmodels, etc.). One thing that does happen is occasional version incompatibility between the various packages, which can be frustrating. If you trust your judgment, this situation is no different than with any software. *Caveat emptor*.

We hope to convince you that for many purposes, the benefits of Python greatly outweigh the difficulties.

1.4 When to Use Python?

There are a few common use cases for Python:

- You want access to methods that are newer or more powerful than available elsewhere. Many Python users start for exactly that reason; they see a method in a journal article, conference paper, or presentation, and discover that the method is available in Python.
- You need to run an analysis many, many times. This is how one author (Chris) started his statistical programming journey; for his dissertation, he needed to bootstrap existing methods in order to compare their typical results to those of a new machine learning model.
- You need to apply an analysis to multiple datasets. Because everything is scripted, Python is great for analyses that are repeated across datasets. It even has tools available for automated reporting.
- You need to develop a new analytic technique or wish to have perfect control and insight into an existing method. For many statistical procedures, Python is easier to code than other programming languages.
- Your manager, professor, or coworker is encouraging you to use Python. We've influenced students and colleagues in this way and are happy to report that a large number of them are enthusiastic Python users today.

By showing you the power of Python, we hope to convince you that your current tools are *not* perfectly satisfactory. Even more deviously, we hope to rewrite your expectations about what *is* satisfactory.

1.5 Using This Book

This book is intended to be *didactic* and *hands-on*, meaning that we want to teach you about Python and the models we use in plain English, and we expect you to engage with the code interactively in Python. It is designed for you to type the commands as you read. (We also provide code files for download from the book's web site; see Sect. 1.5.3 below.)

1.5.1 About the Text

Python commands for you to run are presented in code blocks representing samples, like this:

```
In [1]: print('Hello World')
```

```
Hello World
```

The code is formatted as found in `Notebooks`, which we introduce in Chap. 2. Briefly, notebooks are interactive coding environments that are commonly used by Python programmers, particularly for data analysis, but for many other applications as well. Notebooks are our recommended interface for learning data analysis in Python (See Sect. 2.1 for more info).

We describe these code blocks and interacting with Python in Chap. 2. The code generally follows the PEP 8 Style Guide for Python (available at https://www.python.org/dev/peps/pep-0008/) except when we thought a deviation might make the code or text clearer. (As you learn Python, you will wish to make your code readable; the guide is very useful for code formatting.)

When we refer to Python commands or data in the text outside of code blocks, we set the names in monospace type like this: `print()`. We include parentheses on function names to indicate that they are functions (i.e. commands that reference a set of code), such as the `open()` function (Sect. 2.4.8), as opposed to a variable such as the `store_df` dataset (Sect. 2.4).

When we introduce or define significant new concepts, we set them in italic, such as *vectors*. Italic is also used simply for *emphasis*.

We teach the Python language progressively throughout the book, and much of our coverage of the language is blended into chapters that cover marketing topics and statistical models. In those cases, we present crucial language topics in *Language Brief* sections (such as Sect. 3.2.1). To learn as much Python as possible, you'll need to read the Language Brief sections even if you only skim the surrounding material on statistical models.

Some sections cover deeper details or more advanced topics, and may be skipped. We note those with an asterisk in the section title, such as *Learning More**.

1.5.2 About the Data

Most of the datasets that we analyze in this book are *simulated* datasets. They are created with Python code to have a specific structure. This has several advantages:

- It allows us to illustrate analyses where there is no publicly available marketing data. This is valuable because few firms share their proprietary data for analyses such as segmentation.
- It allows the book to be more self-contained and less dependent on data downloads.
- It makes it possible to alter the data and rerun analyses to see how the results change.
- It lets us teach important Python skills for handling data, generating random numbers, and looping in code.
- It demonstrates how one can write analysis code while waiting for real data. When the final data arrive, you can run your code on the new data.

We recommend working through the data simulation sections where they appear; they are designed to teach Python and to illustrate points that are typical of marketing data. However, when you need data quickly to continue with a chapter, it is available for download as noted in the next section and again in each chapter.

Whenever possible you should also try to perform the analyses here with your own datasets. We work with data in every chapter, but the best way to learn is to adapt the analyses to other data and work through the issues that arise. Because this is an educational text, not a cookbook, and because Python can be slow going at first, we recommend to conduct such parallel analyses on tasks where you are not facing urgent deadlines.

At the beginning, it may seem overly simple to repeat analyses with your own data, but when you try to apply an advanced model to another dataset, you'll be much better prepared if you've practiced with multiple datasets all along. The sooner you apply Python to your own data, the sooner you will be productive in Python.

1.5.3 Online Material

This book has an online component. In fact, we recommend using Colab (see Sect. 2.1.1) for its ease of setup, in which case your code will live and run online.

There are three main online resources:

- An information website: https://python-marketing-research.github.io
- A Github repository: https://github.com/python-marketing-research/python-marketing-research-1ed
- The Colab Github browser: https://colab.sandbox.google.com/github/python-marketing-research/python-marketing-research-1ed

The website includes links to those other sources, as well as any updates or news.

The Github repository contains all the data files, notebooks, and function code.

The data files can be downloaded directly into Python using the `pandas.read_csv()` command (you'll see that command in Sect. 2.6.2, and will find code for an example download in Sect. 3.1). Links to online data are provided in the form of shortened `bit.ly` links to save typing. The data files can be downloaded individually or as a zip file from the repository (https://bit.ly/PMR-all-data).

The notebooks can be downloaded to be run locally using Jupyter (see Sect. 2.1.3). The notebooks can be browsed directly from Colab and easily run using the Colab Github browser (https://colab.sandbox.google.com/github/python-marketing-research). See Chap. 2 for more information.

Note that while we make the notebooks available, we recommend that you use them sparingly; you will learn more if you type the code and create the datasets by simulation as we describe.

In many chapters we create functions that we will then use in later chapters. Those code files are in the Github repository, in the python_marketing_research_ functions directory, and can be download from there to run. However, a far simpler way to access that code is to install the code using pip. See Sect. 2.4.9 for details.

1.5.4 When Things Go Wrong

When you learn something as complex as Python or new statistical models, you will encounter many large and small warnings and errors. Also, the Python ecosystem is dynamic and things will change after this book is published. We don't wish to scare you with a list of concerns, but we do want you to feel reassured about small discrepancies and to know what to do when larger bugs arise. Here are a few things to know and to try if one of your results doesn't match this book:

- **With Python**. The basic error correction process when working with Python is to check everything very carefully, especially parentheses, brackets, and upper- or lowercase letters. If a command is lengthy, deconstruct it into pieces and build it up again (we show examples of this along the way).
- **With packages** (add-on libraries). Packages are regularly updated. Sometimes they change how they work, or may not work at all for a while. Some are very stable while others change often. If you have trouble installing one, do a web search for the error message. If output or details are slightly different than we show, don't worry about it. The error "`ImportError: No module named ...`" indicates that you need to install the package (Sect. 2.4.9). For other problems, see the remaining items here or check the package's help file (Sect. 2.4.11).
- **With Python warnings and errors**. A Python "warning" is often informational and does not necessarily require correction. We call these out as they occur with our code, although sometimes they come and go as packages are updated. If Python gives you an "error," that means something went wrong and needs to be corrected. In that case, try the code again, or search online for the error message. Another very useful tool is adding `print()` statements to print the values of variables referenced in the error or warning; oftentimes a variable having an unexpected value offers a clue to the source of the problem.
- **With data**. Our datasets are simulated and are affected by random number sequences. If you generate data and it is slightly different, try it again from the beginning; or load the data from the book's website (Sect. 1.5.3).
- **With models**. There are three things that might cause statistical estimates to vary: slight differences in the data (see the preceding item), changes in a package that lead to slightly different estimates, and statistical models that employ random sampling. If you run a model and the results are very similar but slightly different, you can assume that one of these situations occurred. Just proceed.
- **With output**. Packages sometimes change the information they report. The output in this book was current at the time of writing, but you can expect some packages will report things slightly differently over time.
- **With names that can't be located**. Sometimes packages change the function names they use or the structure of results. If you get a code error when trying to extract something from a statistical model, check the model's help file (Sect. 2.4.11); it may be that something has changed names.

Our overall recommendation is this. If the difference is small—such as the difference between a mean of 2.08 and 2.076, or a p-value of 0.726 vs. 0.758—don't worry too much about it; you can usually safely ignore these. If you find a large difference—such as a statistical estimate of 0.56 instead of 31.92—try the code block again in the book's code file (Sect. 1.5.3).

1.6 Key Points

At the end of each chapter we summarize crucial lessons. For this chapter, there is only one key point: if you're ready to learn Python, let's get started with Chap. 2!

Chapter 2
An Overview of Python

2.1 Getting Started

In this chapter, we cover just enough of Python to get you going. If you're new to programming, this chapter will get you started well enough to be productive and we'll call out ways to learn more at the end. Python is a great place to learn to program because its syntax is simpler and it has less overhead (e.g. memory management) than traditional programming languages such as Java or C++. If you're an experienced programmer in another language, you should skim this chapter to learn the essentials.

We recommend you work through this chapter *hands-on* and be patient; it will prepare you for marketing analytics applications in later chapters.

There are a few options for how to interact with and run Python, which we introduce in the next few sections.

2.1.1 Notebooks

Notebooks are the standard interface used by data scientists in Python. The notebook itself is a document that contains a mix of code, descriptions, and code output. The document is created and managed using a Notebook app, which is an application that includes a browser app that renders notebook documents, along with a "computational engine" which is a server that inspects and runs code (also called a "kernel"). You use a browser to connect to that server and run Python code in *cells* of the notebook, with output, when present, being printed from each cell. These notebooks allow figures to be embedded, enabling interleaved code, tables, and figures in a single document.

A common workflow is to use a notebook to explore a new dataset and prototype an analysis pipeline. A clean, streamlined version of that pipeline can then be put in another notebook and shared or into a script to be run regularly, or even moved into production code.

Google Colaboratory

The easiest way to get started in Python, and the way that we used in writing the book, is to use Google Colaboratory ("Colab") notebooks. These are free hosted Python notebooks. The notebooks themselves are saved by default in a Google Drive (a cloud storage drive), but can also be saved to Github or downloaded as .ipynb files.

The Python installation running in Colab includes most of the scientific Python libraries that we will use throughout the book. Additional libraries can be installed using the *pip* or *apt* package management systems (see Sect. 2.4.9).

To get started using Colab, go to https://colab.research.google.com/. The initial landing page will be a "Getting started" notebook. To create a new notebook, if you are already viewing an existing notebook, go to the menu bar, open the *File* menu and select *New Notebook*. On subsequent visits, a "Recent notebooks" panel will be displayed when the site is visited, and clicking "New Notebook" will allow you to do so.

If you prefer to run Colab locally, it can also run locally using Jupyter (see Sect. 2.1.3). Visit https://research.google.com/colaboratory/local-runtimes.html for more information.

© Springer Nature Switzerland AG 2020

J. S. Schwarz et al., *Python for Marketing Research and Analytics*, https://doi.org/10.1007/978-3-030-49720-0_2

2.1.2 Installing Python Locally

If you would rather not use a cloud-based system, you can install Python locally.

If you use Linux or Mac OS X, it is likely that Python is already installed. You can check this using the Terminal application to access the command line. Terminal can found in the Applications folder on Mac OS X. On graphical Linux, it is usually prevalent in the Applications explorer, but will sometimes be under "Administration" or "Utilities." Open a Terminal window and type `which python` to check. The command `python --version` will return the version.

All of the code in this book was written and tested using Python version 3.6.7. We recommend using Python 3 rather than Python 2. For the purposes of this book the differences are minor, but there is code that will not run properly in Python 2. Python 2 lost official support on January 1, 2020 (Peterson 2008–2019) and many important libraries dropped Python 2 support long ago (e.g. the `pandas` package stopped support Python 2 on December 31, 2018).

If you don't already have Python 3 installed, the most straightforward way to install Python and all the necessary libraries is using (Anaconda, Inc. 2019) https://www.anaconda.com/. The benefit of using Anaconda rather than a manual install is that it includes all of the libraries that are commonly used in data science applications of Python (see Sect. 2.4.9). Anaconda has a straightforward installation process for Windows, Mac, and Linux.

If you already have Python 3 installed, you can use that, but unless you already have all of the scientific Python libraries, we still recommend installing Anaconda since it includes all of the necessary libraries and tools. Alternatively, you could manually install those libraries (see Sect. 2.4.9).

2.1.3 Running Python Locally

Command Line

If you open a Terminal (Linux/Mac) or Command window (Windows) and type `python`, you will start running Python on the command line in interactive mode. From there, you can run any Python commands that you like. You could perform analyses directly in the command line. However, such a process would be frustrating and not reusable (the command history may not persist across sessions). Better is to save your work so it can be easily modified and repeated.

Scripts

Python code can be written to a file, which is customarily given a `.py` file extension. That file can be run from the command line with the syntax `python <path/to/file>`. For example, we might write code that analyzes monthly sales numbers and call it `monthly_sales.py`, we could run it with the command `python monthly_sales.py`. This file is generally referred to as a *script*.

Scripts are often used when you want to repeatedly run an analysis and generate the same output each time, such as running a monthly or daily analysis. However, they are not necessarily the best development environment for data science applications, as they do not enable interactive exploration. Additionally, any data will need to be loaded into memory each time the script is run, which can slow down development especially if the dataset is large and takes time to load into memory.

Local Notebooks

We have already introduced Google Colaboratory notebooks, which can be run on a free cloud virtual machine instance. But notebooks can also be run locally using Jupyter (Kluyver et al. 2016). Jupyter is included in Anaconda. A Jupyter notebook server can be started by running `jupyter notebook` in the terminal. This will start the server and also launch a browser window to the server overview page, from which you can see any existing notebooks in the current directory or create a new directory. Jupyter supports not only Python but many other programming languages. A local Jupyter runtime can also run Google Colab notebooks. Visit https://jupyter.org for more information.

A Note About Notebooks

As may be clear already, we really like notebooks as tools for analyzing data. Why do we like them so much? The main reason is that they function as *self-contained end-to-end analysis documents*.

When first examining a new dataset, the first step is a series of exploratory analyses, which help to understand the nature of the data. When you perform those exploratory analyses in a notebook, you can always come back to the exact set of steps you performed and see the output at each step. You can annotate each of those steps as well, to make your logic explicit.

Oftentimes, an exploratory analysis like this is not saved, especially in environments where it is tedious to do so (e.g. having to write out the steps in a document or copy over to a script). But in a notebook this exploratory analysis is saved *de facto* and we find ourselves regularly returning to our initial exploratory analysis notebooks even when we have a finalized analysis notebook, to remember the directions we did not yet fully explore.

When an analysis is complete, its associated notebook documents the entire analysis, including data import, data transformation, data summarization, statistical testing or model building, table generation, figure production, and data export. You could write a script that performs the same functions, without the compiled output that a notebook shows. But how useful would that script be to a non-technical stakeholder? Most likely, it would not be very helpful because they would have to read and interpret the code directly. Yet a notebook contains the relevant output along with its context. Now consider a technical colleague. If you share only the output files with them, they will be interested to see how they were generated. You might share a script along with data files, that would require the colleague to rerun all the steps, which may be complex and time-consuming. A notebook solves that problem by combining the code with its results for immediate reference.

Also a notebook is easy to read. We return to past analyses regularly, and having them in notebook form makes it much simpler to find particular results and understand exactly what we did.

Integrated Development Environments

Integrated development environments, generally referred to as IDEs, are applications that simplify development by bringing multiple functions into a single application, such as an interactive command prompt, a file browser, an error console, figure windows, etc. RStudio (2019) is a very popular IDE among R users, and it also supports Python and is available in Anaconda. Some companies may have questions about RStudio's Affero General Public License (AGPL) terms because it poses questions for those who create proprietary code. If relevant, ask your technology support group if they allow AGPL open source software.

A Python-specific IDE is the Spyder IDE, also included with Anaconda (under the less contentious MIT License). The Pycharm IDE is another option. Many general purpose IDEs also work well for Python, including Vim, Emacs, Visual Studio Code, and others. If you are an experienced programmer and already have a development pipeline, chances are it can be easily modified to work with Python in a data science application.

In this book, we assume usage of notebooks, and we encourage everyone to try notebooks out as we feel like they are extremely powerful tools not only for analysis but also for sharing. However, if you are already comfortable with an IDE of your own choice, it is likely to work with Python and this book.

2.2 A Quick Tour of Python Data Analysis Capabilities

Before we dive into the details of programming, we'd like to start with a tour of a relatively powerful analysis in Python. This is a partial preview of other parts of this book, so don't worry if you don't understand the commands. We explain them briefly here to give you a sense of how a Python analysis might be conducted. In this and later chapters, we explain all of these steps in more detail and many more analyses.

If you are not running in Colab or an Anaconda install, you would first need to install a few libraries, which can be done using `pip` on the command line:

```
pip install pandas seaborn statsmodels
```

or in a Colab or Jupyter notebook using the `!` command, which instructs the notebook to run that as a shell command:

```
In [4]: !pip install pandas

Requirement already satisfied: pandas in /usr/local/lib/python3.6/
 dist-packages
```

Colab has pandas installed by default, so no action was necessary.

A quick note about this format. Throughout the book, we format the code so it matches how it will appear in a notebook. The "In" indicates that this is an input cell: we entered code there and ran it. Output cells will be indicated by "Out" and will be present following an output-generating input cell. The number, "4" in this case, indicates the order in which that cell was run, in this case it was the fourth cell to be run in the notebook (there are some earlier demonstration cells in the notebook).

To run a cell, hit "control-Enter" or "command-Enter" (on PC or Mac, respectively). To add a new cell, you can hit the "+ Code" button or hit "control/command + m" and then "a". You can view all Colab keyboard shortcuts by selecting *Keyboard Shortcuts...* from the *Tools* menu.

Most analyses require one or more libraries in addition to the built-in standard Python libraries. After you install a package once, you don't have to install it again unless there is an update, although in Colab it would need to be reinstalled whenever the runtime is "factory reset," which can be done manually (via the "Runtime" menu) or will happen automatically after a few days.

The `import` command is one we'll see often; it loads a package or module with additional functionality, in this case pandas, a data manipulation and analysis package that we will use extensively throughout the book (McKinney 2010).

Now we load a dataset from this book's website and examine it:

```
In [5]: import pandas as pd
        sat_df = pd.read_csv('http://bit.ly/PMR-ch2')
        sat_df.Segment = sat_df.Segment.astype(pd.api.types.
                                               CategoricalDtype())
        sat_df.head()

Out[5]:    iProdSAT  iSalesSAT Segment  iProdREC  iSalesREC
        0         6          2       1         4          3
        1         4          5       3         4          4
        2         5          3       4         5          4
        ...

In [6]: sat_df.describe()

Out[6]:          iProdSAT   iSalesSAT    iProdREC   iSalesREC
        count  500.000000  500.000000  500.000000  500.000000
        mean     4.130000    3.802000    4.044000    3.444000
        std      1.091551    1.159951    1.299786    1.205724
        ...
        max      7.000000    7.000000    7.000000    7.000000
```

This dataset represents observations from a simple sales and product satisfaction survey. It has 500 (simulated) consumers' answers to a survey with 4 items asking about satisfaction with a product (`iProdSAT`), sales experience (`iSalesSAT`), and likelihood to recommend the product and salesperson (`iProdREC` and `iSalesREC` respectively). Each respondent is also assigned to a numerically coded segment (`Segment`). In the third line of code above, we set `Segment` to be a categorical type variable.

Next we chart the correlation matrix, which automatically omits the categorical `Segment` variable in column 3:

```
In [7]: import seaborn as sns
        sns.heatmap(sat_df.corr())
```

The resulting chart is shown in Fig. 2.1 as a heatmap. The satisfaction items are highly correlated with one another, as are the likelihood-to-recommend items.

Does product satisfaction differ by segment? We compute the mean satisfaction for each segment using the `groupby()` method:

```
In [8]: sat_df.groupby('Segment').iProdSAT.mean()

Out[8]: Segment
        1    3.462963
        2    3.725191
        3    4.103896
        4    4.708075
        Name: iProdSAT, dtype: float64
```

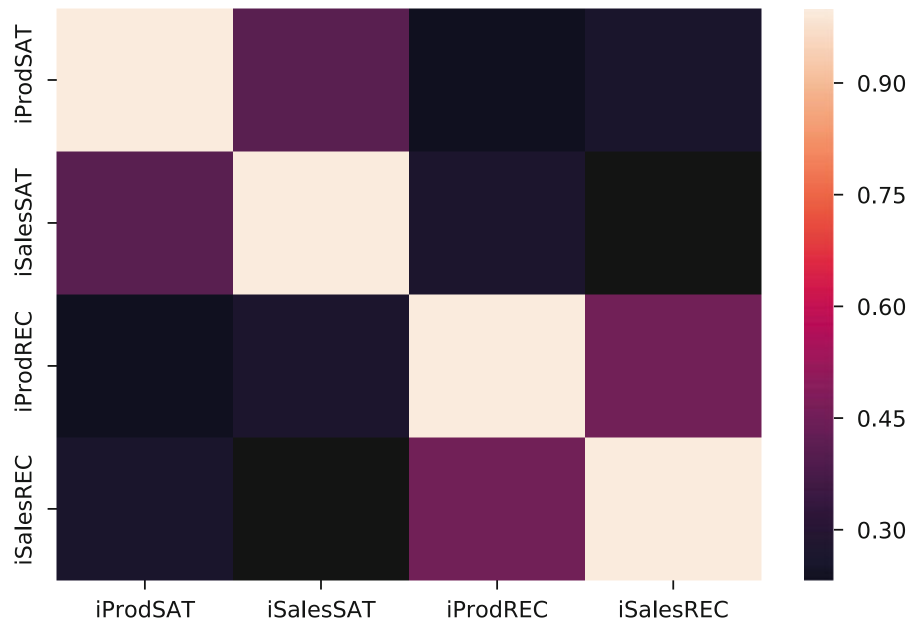

Fig. 2.1 A plot visualizing correlation between satisfaction and likelihood to recommend variables in a simulated consumer dataset, $N = 500$. All items are positively correlated with one another, and the two satisfaction items are especially strongly correlated with one another, as are the two recommendation items. Chapter 4 discusses correlation analysis in detail

Segment 4 has the highest level of satisfaction, but are the differences statistically significant? We perform a oneway analysis of variance (ANOVA) and see in the PR column that satisfaction differs significantly by segment:

```
In [9]: import statsmodels.formula.api as smf
        from statsmodels.stats import anova as sms_anova
        segment_psat_lm = smf.ols('iProdSAT ~ -1 + Segment',
                                  data=sat_df).fit()
        sms_anova.anova_lm(segment_psat_lm)
```

```
Out[9]:            df       sum_sq     mean_sq           F         PR(>F)
        Segment    4.0  8627.850038  2156.962510  2160.66543  3.569726e-312
        Residual 496.0   495.149962     0.998286         NaN            NaN
```

We plot the coefficients and confidence intervals from the ANOVA model to visualize confidence intervals for mean product satisfaction by segment:

```
In [10]: import matplotlib.pyplot as plt
         plt.errorbar(y=segment_psat_lm.params.index,
                      x=segment_psat_lm.params.values,
                      xerr=segment_psat_lm.conf_int()[1].T
                           - segment_psat_lm.params,
                      fmt='ko')
```

The resulting chart is shown in Fig. 2.2. It is easy to see that Segments 1, 2, and 3 differ modestly while Segment 4 is much more satisfied than the others. We will learn more about comparing groups and doing ANOVA analyses in Chap. 5.

In just a few lines of code, we have:

- Imported a dataset
- Done a preliminary inspection of the data
- Checked the correlation between variables
- Compared the mean values by segment
- Run an ANOVA to see if those means vary significantly
- Visualized the confidence intervals of the means to understand the basis of the difference

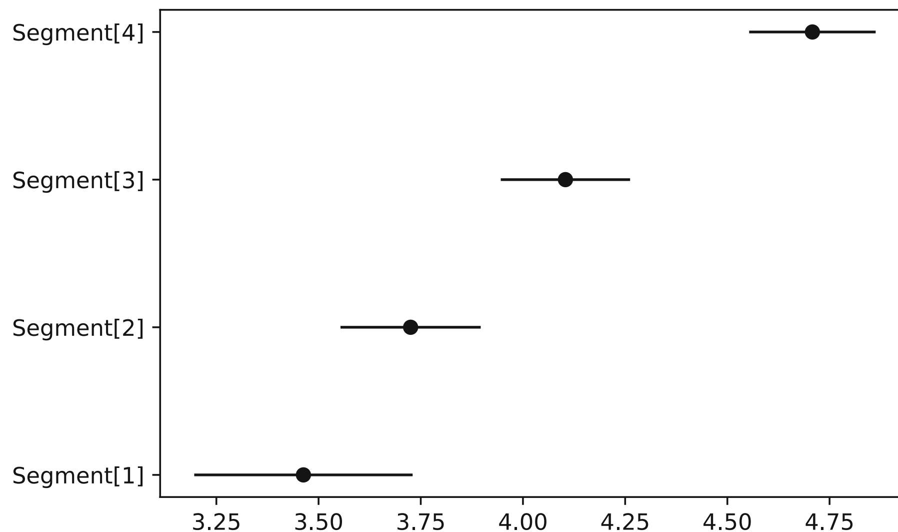

Fig. 2.2 Mean and confidence intervals for product satisfaction by segment. The x-axis represents a Likert rating scale ranging 1–7 for product satisfaction. Chapter 5 discusses methods to compare groups

We ran through this very quickly and did not explain each step; in future chapters we explain all of this (and much more) in detail. Next, we introduce Python as a programming language.

2.3 Basics of Working with Python Commands

Python provides a powerful suite of analytical tools, but fundamentally it is a programming language. The remainder of this chapter serves as an introduction to Python as a programming language. It introduces types and control flow, and then goes onto the data science-focused packages that we will use throughout the book. If you are already familiar with programming in another language, much of this will be very familiar, perhaps except for the data science packages, but if not, it will be helpful to go through these following sections carefully to be sure you understand the fundamentals.

Like most programming languages, Python is *case sensitive*. Thus, `x` and `X` are different:

```
In [11]: x = [1, 23, 6]
         print(x)
```

```
[1, 23, 6]
```

```
In [12]: print(X)
```

```
        NameError: name 'X' is not defined
```

It is helpful to add comments to your code, whether it is in a script or a notebook. These document your thought process and explain what the code is doing. They are essential when sharing code with others, and you will be thankful for them even when reading your own code in the future. The “#” symbol signifies a *comment* in Python, and everything on a line after it is ignored. For example:

```
In [13]: x = [1, 23, 6] # Initialize a list
```

In this book, you don’t need to type any comments; they just make the code more understandable.

The command above defines `x` and ends with a comment. One might instead prefer to comment a whole line:

```
In [14]: # Initialize a list
         x = [1, 23, 6]
```

Our code includes comments wherever we think it might help. As a politician might say about voting, we say *comment early and comment often*. It is much easier to document your code while writing it rather than later.

2.3.1 *Python Style*

Python is a very flexible language, but Python programmers take "style" very seriously. There are heated discussions on what the most "Pythonic" way to solve a problem is! Throughout this book, we aim to conform to style as outlined in the PEP 8 Style Guide for Python Code (https://www.python.org/dev/peps/pep-0008/). The following naming conventions are rules that most Python programmers follow. If you don't follow these rules, your code will run, but other Python programmers may have trouble reading it.

Naming Conventions

Variables are objects that represent a value, such as a number or a string. By convention, all variables in Python are lower cased. If multiple words occur in the variable name, they are separated by an underscore (e.g. `variable_one`), and not camel-cased (e.g. `variableOne`). All variable names must start with an alphabetical character, but can contain any alphanumeric character. Besides underscores, other special characters are excluded.

Class names are upper cased, with each new word being upper cased as well, for example `MyAdder`. See Sect. 2.4.8 for an introduction to Classes.

Single-letter variable names should be avoided unless it is obvious what a name means or it is used as a temporary variable in a single statement (e.g. in a list comprehension, see Sect. 2.4.8). We sometimes skirt that rule in this book with toy example that use `x` or `y` as variables, but for any substantial amount of code you should avoid single-letter variable names.

Line Width

The maximum line width in Python code files should be 79 characters. In this book, we do not exceed 70 characters (due to the margin width). The reason for this rule is that many code editors wrap lines that are longer than 79 characters, which makes it extremely hard to read.

This rule sometimes requires breaking up statements across multiple lines. This is preferably done using parentheses or other brackets, because once they are opened they will extend across lines or, if necessary, using the '\' character. We will see examples of this throughout the book.

White Space

White space is **very** important in Python. Unlike many other language, white space is meaningful. We go into depth on this topic in Sect. 2.4.8.

2.4 Basic Types

Python has several built-in data types that are important to understand. You can check the Python documentation at https://docs.python.org/3/library/stdtypes.html for more detailed information on each type.

2.4.1 *Objects and Type*

Nearly all of the entities in Python are *objects*. From numbers and strings to functions and classes, these are all objects. Python is a *weakly typed* or *dynamically typed* language, which means a few things.

One, that after declaring an object, not only can its value be changed, but its type as well. In some languages the following would lead to a `TypeError`:

```
a = 3   # Declare a as an int (integer number)
a = 'b' # Reassign a with a string value (text)
```

In Python, such a type change is allowed. Another feature of Python is that many functions don't check the type of their inputs, which can lead to unexpected behavior. Additionally, many basic operators (e.g. the + operator) are *overloaded*, meaning that they function differently depending on the type of their inputs. These two facts combined can lead to some very tricky bugs.

Overall, Python's weak typing makes the language more flexible and easier to code in than a strongly typed language like Java, but it is worth being aware of the potential pitfalls of dynamic typing.

Now to introduce some of the basic types in Python.

2.4.2 Booleans

A boolean (or 'bool') can have only one of two values `True` or `False`.

Bools are often produced from comparisons:

```
In [15]: 1 == 1
Out[15]: True
In [16]: 1 < 2
Out[16]: True
In [17]: 1 == 2
Out[17]: False
```

Bools can also be compared using the `and`, `or`, and `not` operators:

```
In [18]: x = True
         y = False
         x or y
Out[18]: True
In [19]: x and y
Out[19]: False
In [20]: x and not y
Out[20]: True
```

Bools are extremely important for procedural control (see Sect. 2.4.10) and also for indexing dataframes.

2.4.3 Numeric Types

Python has three built-in numeric types: *int*, *float*, and *complex*. Given that this book is focused on data analysis, numeric types are of paramount importance.

A *float* is the Python type for floating-point numbers, which can represent all real numbers (although there are caveats with binary representation of floating point numbers, see https://docs.python.org/3/tutorial/floatingpoint.html and Bush 1996).

Floats can represent decimal values, unlike *ints*, which only represent integers. ints are more memory efficient, so it is best practice to use int types when possible (i.e. whenever representing integer, non-decimal values).

Complex objects represents complex numbers, those that include an imaginary component. We won't make use of them in this book.

Simple arithmetic operations are supported for numeric types, such as addition using the + operator:

```
In [21]: x = 2
         y = 4
         x + y
Out[21]: 6
```

Division using the / operator:

```
In [22]: w = x/y
         w

Out[22]: 0.5

In [23]: type(w)

Out[23]: float
```

Exponentiation using the ** operator:

```
In [24]: x ** y

Out[24]: 16
```

And other operations as well. Note that in the division operation, despite the fact that both numerator and denominator are ints, the output is of type float. This behavior differs between Python 3 and Python 2 (Python 2 would return an `int` with the decimal component truncated; that was a source of many bugs!)

Similarly, if the inputs to an operation include both ints and floats, the output will be a float:

```
In [25]: z = 3.2
         type(x * z)

Out[25]: float
```

2.4.4 *Sequence Types*

Python contains three *sequence* types, each of which is an ordered array of objects.

Lists

Lists are ordered, mutable sequences of objects. They are defined with square brackets, `[]`:

```
In [26]: x = [0, 1, 2, 3, 4, 5]
         y = ['a', 'b', 'c']
```

Lists can be added together, which concatenates them:

```
In [27]: x + y

Out[27]: [0, 1, 2, 3, 4, 5, 'a', 'b', 'c']
```

Objects can be added to the end of a list using the `append()` method:

```
In [28]: x.append('r')
         x

Out[28]: [0, 1, 2, 3, 4, 5, 'r']
```

Unlike many other languages, appending to a list in Python is memory efficient, so there is no need to preallocate the list if you are generating it by iterating through some other object.

Note that lists can contain a mix of types, e.g. ints and strings in the past two examples.

The built-in `len()` function returns the length of lists (as well as other objects):

```
In [29]: len(x)

Out[29]: 7
```

Indexing

Indexing in Python is extremely powerful. Note that lists and other sequence objects in Python are *zero-indexed*, which means that the index starts at 0, not 1. So the *first* item has index *0* and the *last* item has index *length minus 1*.

In the list `x` that we already defined, the second value is `1`, which we access with the index `1`:

```
In [30]: x[1]

Out[30]: 1
```

A range of values can be indexed using the `:` operator:

```
In [31]: x[2:4]

Out[31]: [2, 3]
```

Note that in Python, the lower bound is *inclusive* whereas the upper bound is *exclusive*. The command `x[2:4]` grabs the items with indices 2 and 3 (which, in this case also corresponds to the values).

If you want to start indexing from the beginning of the list, a starting number does not need to be specified. For example, to retrieve the first two elements of `x`:

```
In [32]: x[:2]

Out[32]: [0, 1]
```

Similarly, if you want to index all the way to the end of the list, the final index does not need to be specified:

```
In [33]: x[1:]

Out[33]: [1, 2, 3, 4, 5, 'r']
```

Negative indices are relative to the end of the list. So, for example, to retrieve the last two elements of `x` you can grab everything from the $-2th$ element:

```
In [34]: x[-2:]

Out[34]: [5, 'r']
```

This makes for somewhat cleaner, more readable code than in many other languages, where the length of the list would need to be calculated to get the same functionality.

As mentioned before, lists are mutable. This means that they can be appended to, but also that the value at any index can be changed:

```
In [35]: x[2] = 'freeze'
         x

Out[35]: [0, 1, 'freeze', 3, 4, 5, 'r']
```

Tuples

Tuples are similar to lists with one major caveat: they are immutable.

Tuples are defined with parenthetical brackets, `(,)`:

```
In [36]: z = (7, 8, 9)
```

Tuples are indexed just like lists:

```
In [37]: z[1]

Out[37]: 8
```

Attempting to modify a tuple leads to a `TypeError`:

```
In [38]: z[1] = 'boil'
```

```
TypeError: 'tuple' object does not support
  item assignment
```

We will not make extensive use of tuples in this book, but it is good to be aware of them. Since tuples are immutable, they can be *hashed*, which enables them to be used in sets and dictionaries, which we introduce later in the chapter.

Ranges

Ranges are *immutable* sequences of *numbers*, most commonly used for looping via `for` loops.

A range is defined by positional arguments `start`, `stop`, and `step`. Only `stop` is required. If only `stop` is provided, the range will start at 0 and increment by 1 up to that value:

```
In [39]: range(10)

Out[39]: range(0, 10)

In [40]: list(range(10))

Out[40]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

A few things to note. One, that a `range` must be cast to a list in order to view its contents (note: in Python 2 this is not true, but rather the output of `range()` is a `list` rather than a `range` object).

Secondly, that much like indexing, the upper limit is *exclusive*, that is, the sequence stops at the last value prior to the `stop` argument value. This makes sense, as given that lists are zero-indexed, it means that the list will have length equal to `stop`.

And again, the `start` argument is *inclusive*:

```
In [41]: list(range(2,12))

Out[41]: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

If `step` is specified, the range will start at `start`, increment by `step` while the value remains less than `stop`:

```
In [42]: list(range(2, 12, 2))

Out[42]: [2, 4, 6, 8, 10]
```

2.4.5 Text Type: String

Python has a single type for specifying text: `str` or *strings*. Strings can be specified by `'single'`, `''double''`, or `'''triple'''` quotes. Single and double quotes behave identically; triple quotes act across lines.

Strings share some characteristics with lists, such as concatenation through `+`:

```
In [43]: x = 'Hello'
         y = "World"
         x+y

Out[43]: 'HelloWorld'
```

And indexing using `[]`:

```
In [44]: x[3:]

Out[44]: 'lo'
```

Strings also have many string-specific methods as well such as case modification:

```
In [45]: x.upper()

Out[45]: 'HELLO'
```